Azure, ASP.NET Core, .NET MAUI, Blazor

CODE MAGAZINE

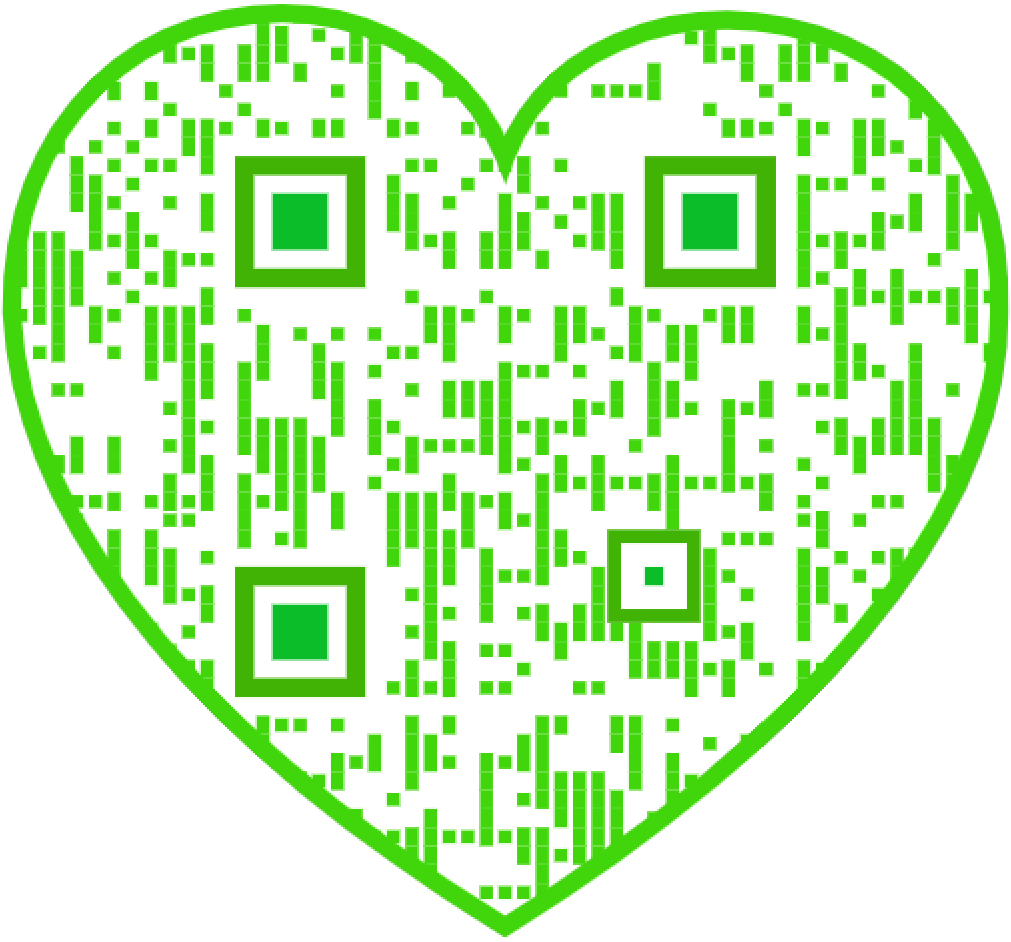
MAY
JUN
2025

codemag.com - THE LEADING INDEPENDENT DEVELOPER MAGAZINE - US $ 8.95 Can $ 11.95

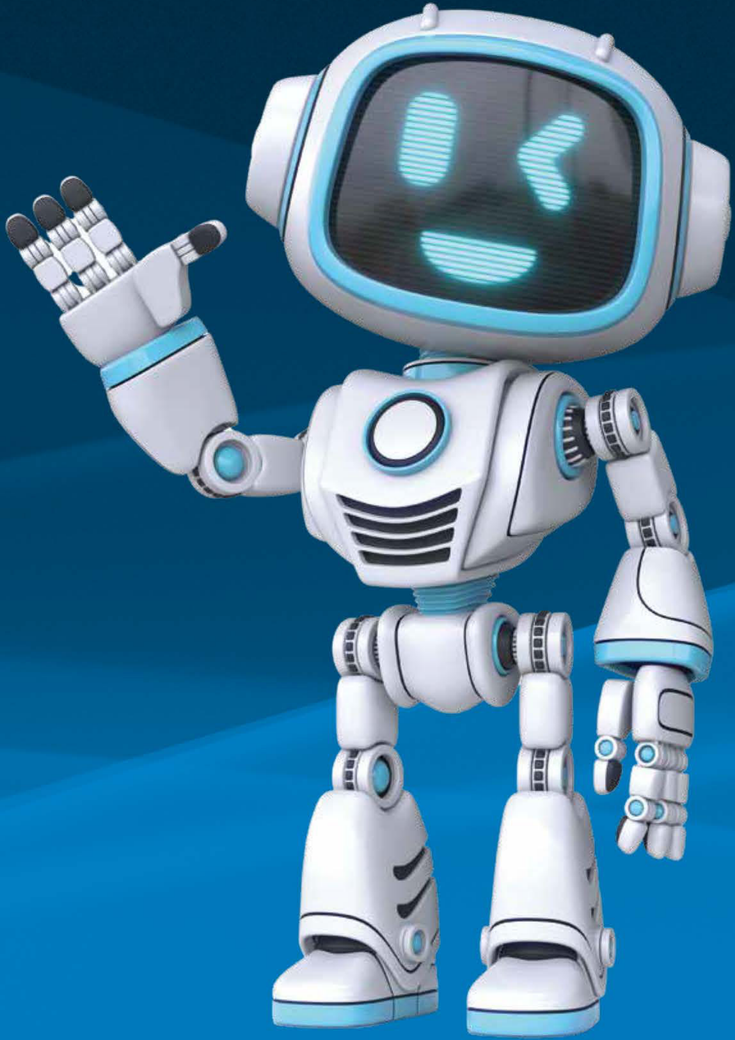
Creating Serverless Functions

Building Blazor Web Apps

Riding the Azure Service Bus



CODE CONSULTING



©shutterstock

ARE YOU WONDERING HOW ARTIFICIAL INTELLIGENCE CAN BENEFIT YOU TODAY?

EXECUTIVE BRIEFINGS

Are you wondering how AI can help your business? Do you worry about privacy or regulatory issues stopping you from using AI to its fullest? We have the answers! Our Executive Briefings provide guidance and concrete advise that help decision makers move forward in this rapidly changing Age of Artificial Intelligence and Copilots!

We will send an expert to your office to meet with you. You will receive:

1. An overview presentation of the current state of Artificial Intelligence.
2. How to use AI in your business while ensuring privacy of your and your clients' information.
3. A sample application built on your own HR documents – allowing your employees to query those documents in English and cutting down the number of questions that you and your HR group have to answer.
4. A roadmap for future use of AI catered to what you do.

AI-SEARCHABLE KNOWLEDGEBASE AND DOCUMENTS

A great first step into the world of Generative Artificial Intelligence, Large Language Models (LLMs), and GPT is to create an AI that provides your staff or clients access to your institutional knowledge, documentation, and data through an AI-searchable knowledgebase. We can help you implement a first system in a matter of days in a fashion that is secure and individualized to each user. Your data remains yours! Answers provided by the AI are grounded in your own information and is thus correct and applicable.

COPILOTS FOR YOUR OWN APPS

Applications without Copilots are now legacy!

But fear not! We can help you build Copilot features into your applications in a secure and integrated fashion.

CONTACT US TODAY FOR A FREE CONSULTATION AND DETAILS ABOUT OUR SERVICES.

codemag.com/ai-services
832-717-4445 ext. 9 • info@codemag.com

Features

8 Front-End Development Trends

Designing a front-end isn't all dialog boxes and clickable buttons. Sahil shares how semantic HTML, dialog tags, HTML forms, and vector graphics make interacting with your app better for you and your users.

Sahil Malik

16 Exploring .NET MAUI: Popups, Messages, and Data Validation

In this sixth entry in his series on MAUI, Paul teaches you how to use data annotation attributes, generic code, and popup messages to get your users where they want to go.

Paul Sheriff

34 Get on the Bus— The Azure Service Bus

Microsoft's Azure Service Bus is for developing message-based applications. Matthew explains some basics, like configuring queues, sending and receiving messages, and how to handle exceptions and attacks from the bad guys.

Matthew Hess

46 Securing ASP.NET Core Blazor Applications

Blazor helps build interactive web applications using C# and .NET in a modular component model. Joydip shows you how flexible it can be when you need a rich interactive web user interface.

Joydip Kanjilal

68 Unlock Serverless Azure Functions

One of the exciting developments in cloud computing is its serverless architecture. Vaibhav explains how, without the overhead of managing servers, monitoring infrastructure, and other operational concerns, you can focus on writing great code and getting it to your users faster.

Vaibhav Gujral

Departments

6 Editorial

32 Advertisers Index

74 Code Compilers

We thank these sponsors who have helped us fullfill our goal of empowering LOB developers.
Their support is the foundation of our silver anniversary.

dtSearch

IRONSOFTWARE

CODE STAFFING

next GenAI CONFERENCE

US subscriptions are US $29.99 for one year. Subscriptions outside the US pay $50.99 USD. Payments should be made in US dollars drawn on a US bank. American Express, MasterCard, Visa, and Discover credit cards are accepted. Back issues are available. For subscription information, send e-mail to subscriptions@codemag.com or contact Customer Service at 832-717-4445 ext. 9.

Subscribe online at www.codemag.com

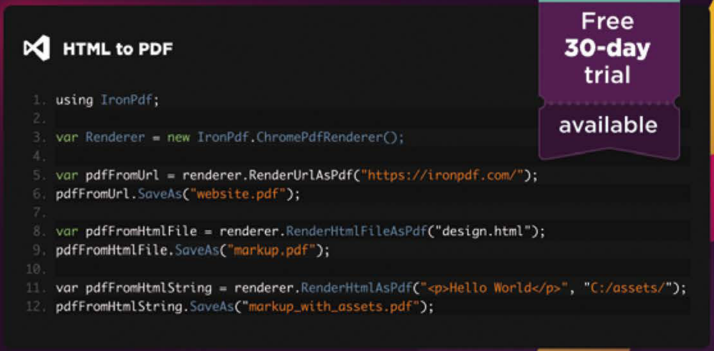
CODE Component Developer Magazine (ISSN # 1547-5166) is published bimonthly by EPS Software Corporation, 6605 Cypresswood Drive, Suite 425, Spring, TX 77379 U.S.A. POSTMASTER: Send address changes to CODE Component Developer Magazine, 6605 Cypresswood Drive, Suite 425, Spring, TX 77379 U.S.A.

IRONPDF for .NET

C# PDF Library that prioritizes accuracy, ease of use, and speed.

IronPDF is the leading C# PDF library for generating & editing PDFs. Its user friendly API allows developers to rapidly deliver professional, high quality PDFs from HTML in .NET projects.

When you need your PDF to look like HTML, fast.



HTML to PDF

```
using IronPdf;

var Renderer = new IronPdf.ChromePdfRenderer();

var pdfFromUrl = renderer.RenderUrlAsPdf("https://ironpdf.com/");
pdfFromUrl.SaveAs("website.pdf");

var pdfFromHtmlFile = renderer.RenderHtmlFileAsPdf("design.html");
pdfFromHtmlFile.SaveAs("markup.pdf");

var pdfFromHtmlString = renderer.RenderHtmlAsPdf("<p>Hello World</p>", "C:/assets/");
pdfFromHtmlString.SaveAs("markup_with_assets.pdf");
```

Free
30-day
trial
available

Downloads 13M

`PM > Install-Package IronPdf`

.NET Mac docker Azure aws Visual Studio nuget R#

Designed for **C#, F#, & VB.NET** running on **.NET 9, 8, 7, 6, Core, Standard,** or **Framework.** Available for **.NET, Java, Node.js, and Python.**

Create PDFs

Design Perfect PDFs
- Create Blank PDF
- Add Text & Images
- Add Shapes
- Add Headers & Footers
- Add Page Numbers

Full PDF Customization
- Orientation
- Custom Paper Size
- Set PDF Metadata

Standards Compliance
- PDF Versions (1.2 - 1.7)
- PDF/UA (PDF/UA-1)
- PDF/A (1-3 b/a)

PDF Viewing and Printing
- IronPDF Viewer
- Print to a Physical Printer

Convert PDFs

Versatile PDF Conversion
- HTML to PDF
- URL to PDF
- DOCX to PDF
- RTF & MD to PDF
- Image to PDF

Dynamic Web Page to PDFs
- Razor to PDF
- CSHTML to PDF
- ASPX to PDF
- XAML to PDF

Web Accessibility
- TLS Website & System Logins
- Cookies

Tailored PDF Conversion
- Custom Margins
- Set Headers & Footers
- Add Page Numbers
- Grayscale

Refine PDF Layout
- Add a Table of Contents
- Add Page Breaks
- Fit to Paper

Edit PDFs

Edit PDF Objects
- Translate PDF Objects
- Scale PDF Objects
- Remove PDF Objects

Edit Document Text
- Extract PDF Text & Images
- Redact Text
- Find and Replace Text

Enhance PDF Design
- Annotations
- Stamp Texts & Images
- Custom Watermark
- Draw Text & Bitmap
- Draw Line and Rectangle
- Add Headers/Footers
- Add Page Numbers
- Background & Foreground

Organize PDFs

Edit PDF Structure
- Add, Copy, & Delete Pages
- Merge and Split PDFs

Supplementary Organization
- Bookmarks and Outline
- Attachments

Sign and Secure PDFs

Ensure Authenticity
- Digital Signatures (.pfx, .p12)
- Edit & Sign Revision History
- Edit PDF Metadata

Web Assets Support
- Create PDF Forms
- Fill & Edit Existing Forms
- Flatten Form Fields

Document Safety
- Sanitize PDFs
- Set Passwords & Permissions

Others

Support Web Assets
- HTML
- CSS
- JavaScript
- Images
- Fonts
- UTF-8 Character Encoding
- Based URL & Asset Encoding

Performance and Compression
- PDF File Size Compression
- Async & Multithreading
- Custom Logging

> *I've used **IronPDF** on several commercial projects with great results. The HTML-to-PDF conversion just works, and being able to pass a model into a Razor view gives me full control over the output. It really is that simple.*

Milan Jovanović
Microsoft MVP and Tech Blogger

**Backed by global
24/5 live support**

Up and running in 5 minutes, Start Free Trial
www.ironpdf.com

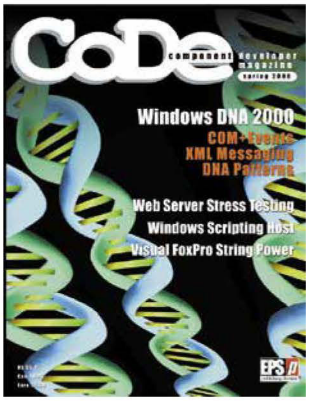
25 Years of CODE Magazine

Welcome to the silver edition of CODE Magazine. Yes! It's been 25 years since the first issue of this magazine, and what a ride it has been. Over the last 25 years, this magazine has covered new, old, and futuristic technologies with one person in mind: you, gentle reader! It is, and always has been, our goal to help the LOB (line of business) developers get their work done. Now, take a journey with us to celebrate some milestones of this trusty publication.

25 years is a lifetime in the software business, and it has been a wild ride for sure. Let's take a look at some of the interesting milestones along the way.

2000: Origin Story

When I was tasked with building this 25th anniversary article, I did what any good author does: I consulted the source material. I opened the www.codemag.com website and went directly to issue #1 in the archive. I was curious what the original topics were and who authored them.

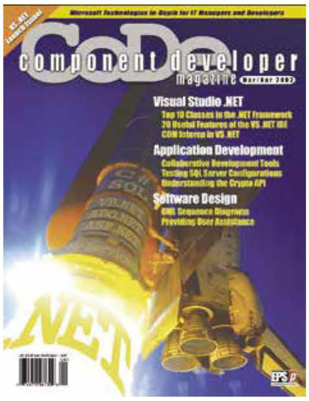

CoDe
component developer magazine
Windows DNA 2000
COM+Events
XML Messaging
DNA Patterns
Web Server Stress Testing
Windows Scripting Host
Visual FoxPro String Power

The first issue, from Spring 2000

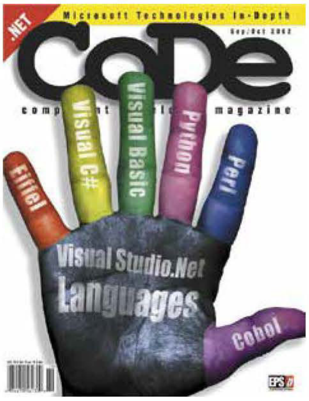
I'm not sure what's more interesting, the list of topics or the list of authors. The "cutting edge" topics included Windows DNA, including COM/COM+, XML, Visual FoxPro, Windows DNA, and two articles on string manipulation. I guess string manipulation has never ceased being an issue in software development. LOL. With publisher Markus Egger's background in the Visual FoxPro community, the roster of authors came from there and consisted of: Markus Egger, Rick Strahl, Steven Black, Yair Alan Griver, Nancy Folsom, Barbara Peisch, and Rod Paddock. I was literally shocked to recall that, yes, I did contribute to the first issue of this magazine and 25 years later, I'm still fortunate enough to call myself a contributor.

2000–2010

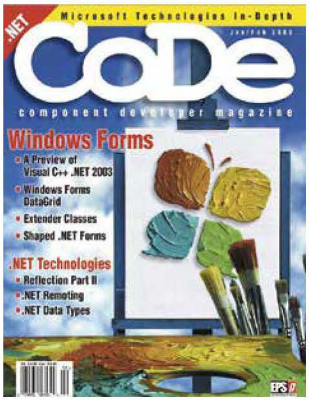
The early 2000s was a great time to be a software developer. We'd just defeated the Y2K dragon and were knee deep in discovering just how powerful web development could be.


component developer magazine
Visual Studio .NET
Application Development
Software Design
.NET

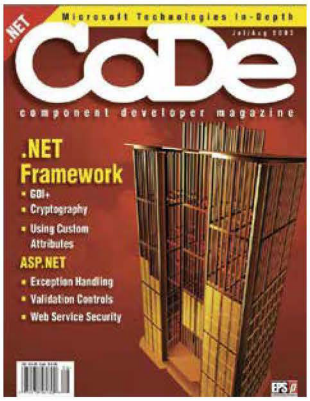
The MarApr 2002 issue


.NET
Microsoft Technologies In-Depth
CoDe
component magazine
Eiffel
Visual C#
Visual Basic
Python
Perl
Visual Studio.Net Languages
Cobol

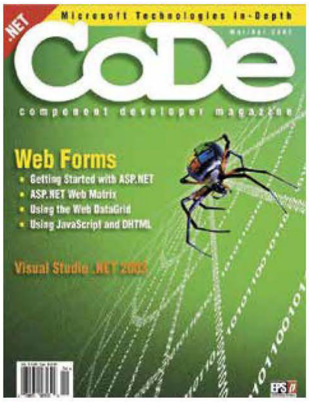
The SeptOct 2002 issue


.NET
Microsoft Technologies In-Depth
CoDe
component developer magazine
Windows Forms
.NET Technologies

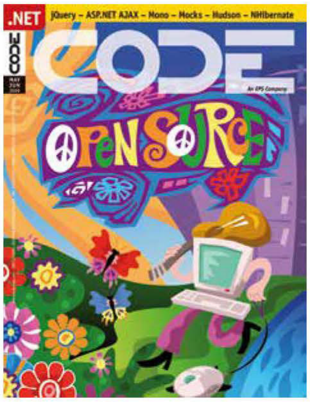
The JanFeb 2003 issue


.NET
Microsoft Technologies In-Depth
CoDe
component developer magazine
.NET Framework
ASP.NET

The JulyAug 2003 issue


.NET
Microsoft Technologies In-Depth
CoDe
component developer magazine
Web Forms
Visual Studio .NET 2003

The MarApr 2003 issue


.NET
jQuery – ASP.NET AJAX – Mono – Mocks – Hudson – NHibernate
CODE
OPEN SOURCE

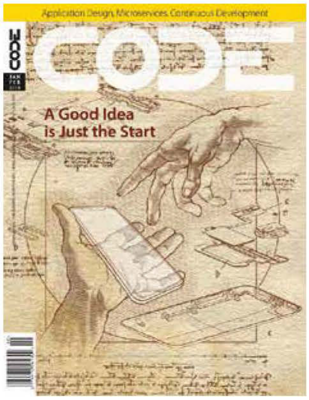
The MayJune 2009 issue

There were numerous developments and milestones in this era.

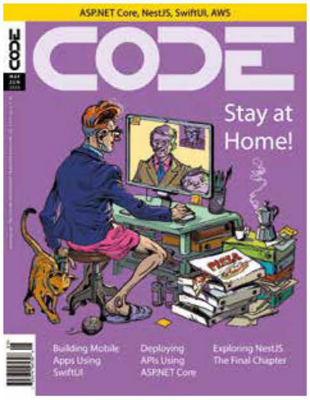
I would venture to say that the 2002 launch of Amazon Web Services (AWS) was probably the biggest development. Their EC2 (Cloud Compute) and S3 (Storage) technologies provide the foundation for literally thousands of applications today. Microsoft would follow Amazon to this space with the launch of Azure in 2008.

Of equal importance, the release of the .NET Framework changed the landscape of development in significant ways. Libraries could be built and shared from multiple programming languages. Different languages could be implemented using the same Common Language Runtime (CLR). Web and windows development were greatly simplified with WinForms and WebForms.

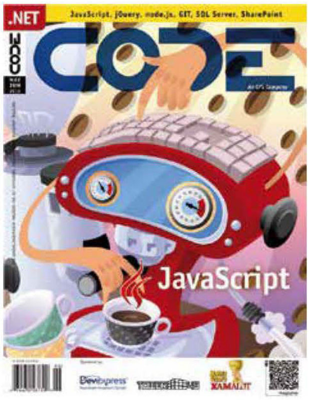
Oh, and let's not forget one small release. In July of 2004, David Heinemeier Hanson (DHH) released his Ruby on Rails project into the world.


Application Design, Microservices, Continuous Development
CODE
A Good Idea is Just the Start

The JanFeb 2018 issue


ASP.NET Core, NestJS, SwiftUI, AWS
CODE
Stay at Home!
Building Mobile Apps Using SwiftUI
Deploying APIs Using ASP.NET Core
Exploring NestJS The Final Chapter

The MayJune 2020 issue


.NET
JavaScript, jQuery, node.js, GIT, SQL Server, SharePoint
CODE
JavaScript

The MayJune 2011 issue

2010–2020

The second decade of the twenty first century was spent discovering the power of distributed computing, adopting open-source development and the creation of the App. We built applications on the foundations of developments from the prior decade.

Start your engines with the release of the V8 JavaScript project. When creating Chrome, Google open-sourced the JavaScript engine. This would go on to be one of the most impactful things Google ever did. JavaScript was being widely adopted, and advances seemed endless.

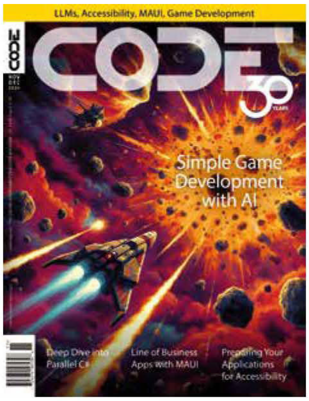
One advance, in particular, would go on to be huge. Sometime around 2010-2011, I was at a conference called TXJS. In one session, a developer showed an application hosting a huge set of sessions on a single instance of a web server being powered by JavaScript. The crowd was in awe. How could you host a web application capable of so many sessions on a single web server with JavaScript as the language? That engine was running this new-fangled framework called Node.js, and that's the rest of the story (sorry Paul Harvey).

The early part of this decade also brought widespread adoption of two technologies that were intertwined like peas and carrots: App development and distributed (cloud) computing. It goes without saying that the release of the iOS and Android operating systems were tectonic changes to the way people interacted with technology. Developers could put their applications into the hands of literally billions of users with the click of a few buttons. This wouldn't be possible without the advent of distributed computing.

2020–2025 and Beyond

The last five years have been in serious overdrive for software developers. Every week (it seems like daily sometimes) brings new advancements/issues. Here are a few highlights:

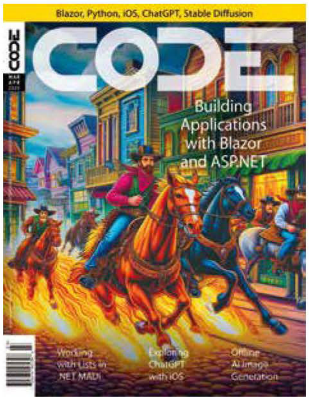
- Containerization with apps like Docker became commonplace
- Distributed computing has become simpler, yet able to build more complex and sophisticated applications
- Open source development became prevalent
- Security is becoming more paramount
- The advent of artificial intelligence (AI) and machine learning (ML)


LLMs, Accessibility, MAUI, Game Development
CODE
30 YEARS
Simple Game Development with AI
Deep Dive into Parallel C#
Line of Business Apps with MAUI
Preparing Your Applications for Accessibility

The NovDec 2024 issue

LangChain, Microservices, AI, Python, C#
CODE
Building AI Pipelines with LangChain
Use CORS in Microservices
Build Game Encounters with AI
Construct Voice-Enabled AI Applications

The JanFeb 2025 issue


Blazor, Python, iOS, ChatGPT, Stable Diffusion
CODE
Building Applications with Blazor and ASP.NET
Working with Lists in .NET MAUI
Exploring ChatGPT with iOS
Offline AI Image Generation

The MarApr 2025 issue

The release of large language models (LLMs) and their associated tooling and techniques will provide a foundation for the next decade of applications. As usual, we'll examine the tools and techniques and try to provide guidance about how you can deploy these applications in your work. We at CODE Magazine will be there with you because we ARE you!

Thanks for 25 years! Here's to 25 more!

Rod Paddock

ONLINE QUICK ID 2505021

Front-End Development Trends

Sahil Malik
www.winsmarts.com
@sahilmalik

Sahil Malik is a Microsoft MVP, INETA speaker, a .NET author, consultant, and trainer.

Sahil loves interacting with fellow geeks in real time. His talks and trainings are full of humor and practical nuggets.

His areas of expertise are cross-platform Mobile app development, Microsoft anything, and security and identity.

Sometimes we hold so much in our projects that we forget to see what's going on in the rest of the world. As developers, especially as full stack developers, it's becoming increasingly challenging to stay up to date with every detail of technology. I feel that everything is becoming a specialized skill. The breadth of knowledge you need to have to call yourself, let's say, a front-end developer is quite expansive and specialized. I thought it would be a good idea to focus an article on the latest trends in front-end development. Yes, we've all heard those rejoinders, like HTML is not a programming language, but let me assure you, it's definitely a specialized skill.

In this article, I'll go through some interesting new trends that have emerged in front-end development.

Semantic HTML

When HTML was first created, it was used to present some basic formatting. You could bold certain text, you could underline certain text, etc. Later, this activity became interactive with forms. Over time, these forms became more complicated and AJAX-based tools really unleashed the possibility of what could be possible in a browser. Then, paired with the additional complexity of security and the fact that accessibility has become extremely important and essential, our plain old HTML friend started clearly showing its age. As a result, HTML had to evolve.

These days, the vast majority of new applications are served through the browser. In fact, sit back and think about it: How many new products that have been launched in the last 10 years that you use on your laptop or desktop are not web applications?

Most of these websites have some common features. They all seem to have a header, paragraphs, navigation footers, etc. For the longest time, we've been trying to twist tags, like div and span, with some clever CSS to act like a header, paragraphs, navigation footers, etc. Semantic HTML changes that.

Semantic HTML is a way of writing HTML that focuses on the meaning of the content rather than just its presentation. It involves using HTML elements that describe the structure and purpose of the content, making it more readable, accessible, and maintainable. Semantic HTML uses elements that provide meaning to the structure of the web page.

Semantic HTML is a standard, as defined by the World Wide Web Consortium (W3C) and maintained by the WHATWG (Web Hypertext Application Technology Working Group). The HTML specification, which includes the definition of semantic HTML elements, is a standard document that outlines the syntax, structure, and semantics of HTML. This specification is maintained by the WHATWG and is widely adopted by web browsers and other HTML parsers.

The use of semantic HTML elements, such as <header>, <nav>, <main>, <section>, <article>, <aside>, <footer>, and others, is mandated by the HTML specification. These elements provide a standardized way to define the structure and meaning of web content, making it easier for browsers, search engines, and other tools to understand and interpret the content.

Although there may be some flexibility in how semantic HTML elements are used, the specification provides clear guidelines on their usage and meaning.

Semantic HTML in practice means that instead of writing code like:

```
<div class=paragraph/>
```

You instead write:

```
<paragraph/>
```

Although seemingly minor, this has huge positive ramifications.

Semantic HTML means improved accessibility. Semantic HTML helps screen readers and other assistive technologies understand the structure and content of the web page, making it easier for people with disabilities to navigate and understand the page.

Sematic HTML translates to better search engine optimization (SEO). Search engines can easily understand the content and structure of the web page, which can improve the page's ranking in search engine results.

Semantic HTML also means easier maintenance and updates. Semantic HTML makes it easy to modify and update the content and structure of the web page, as the HTML elements provide a clear and consistent structure.

Finally, semantic HTML translates into an improved user experience. Semantic HTML can help create a more consistent and intuitive user experience, as the structure and content of the web page are clearly defined.

Let's understand this with an example. **Listing 1** shows a simple web page using non-semantic HTML. Let's read through it. The div class **nav** tells you that this is navigation. But if this was a marine site, maybe it meant navy? What if it was a Native American site; it could mean Navajo. Okay, wait, the next section is a bunch of UL/LIs. From the time I learned HTML way back when, I learned that this is how you create an unordered list. So maybe this is navigation perhaps? Now I read the contents, home, about, contact, and I have a reasonable degree of confidence that this is navigation. And not naval navigation.

But now let's look at the same code expressed in semantic HTML in **Listing 2**. It's pretty clear what it does because the **nav** tag, always means navigation. Not Navajo, naval, or navel. It's website navigation. You can imagine that in more complex pages, this is a huge help to accessibility, search engines, maintainability of code, and if everyone used these standards, it would create a much more consistent user experience.

The Dialog Tag

As a user interacts with your application, you frequently have to interrupt the flow to ask the user a question or present the user with some information. For example, when the user is attempting to do something sensitive, maybe you want to send them an SMS and have the user enter that SMS code. Alternatively, maybe the user is booking a ticket and you want to alert them that the ticket the user is attempting to buy has already been sold out.

Conventionally, we've shown this information via a pop-up window. The typical way of doing this is **window.popup**, as shown in **Listing 3**.

The code example in **Listing 3** creates a button with the ID openPopupBtn. Then it adds an event listener to the button that listens for a click event. When the button is clicked, you use window.open to create a new pop-up window. You specify the URL of the pop-up content (popup-content.html), the name of the pop-up (myPopup), and the features of the pop-up (popupFeatures). You check to see if the pop-up was successfully opened by checking whether the window.open method returned a non-null value.

This sounds simple enough but it has some disadvantages, which the dialog tag intends to address. The <dialog> tag was introduced in HTML5, which was published as a W3C Recommendation on October 28, 2014. However, the <dialog> tag was actually added to the HTML5 specification in 2013.

The window.popup or window.open methods were initially created with good intent, to capture user information in a window, but now they're used frequently to open new tabs in background, frustrating the user with overlays and clickjacking, intending the user to get frustrated and click on things they don't intend to do, such as accepting cookies, increasing local storage quotas, or granting permissions, or any combination thereof. As a result, browsers have evolved to create pop-up blockers and enable them by default, which leads to poorer user experience, buggy applications, and more complicated code.

Secondly, the window.popup window is modeless, i.e., the user is free to interact with the parent window while the pop-up is open. This again requires complex workarounds that are still not fool proof, and lead to poorer user experience.

The window.popup also relies on a new HTML page, and has limited customizability when it comes to styling. You're effectively styling another HTML page, and browsers may choose to show the address bar, really marring your user experience.

Finally, window.popup is not great for accessibility. It requires you to write complex code using manual focus management and ARIA attributes to ensure accessibility.

Listing 1: Non semantic HTML

```
<div class="nav">
  <ul>
    <li><a href="#">Home</a></li>
    <li><a href="#">About</a></li>
    <li><a href="#">Contact</a></li>
  </ul>
</div>

<div class="header">
  <h1>Welcome to My Website</h1>
</div>

<div class="content">
  <p>This is a sample web page.</p>
  <p>Lorem ipsum dolor sit amet..</p>
</div>
```

Listing 2: Semantic HTML

```
<nav>
  <ul>
    <li><a href="#">Home</a></li>
    <li><a href="#">About</a></li>
    <li><a href="#">Contact</a></li>
  </ul>
</nav>

<header>
  <h1>Welcome to My Website</h1>
</header>

<main>
  <p>This is a sample web page.</p>
  <p>Lorem ipsum dolor sit amet..</p>
</main>
```

Listing 3: Using window.popup

```
<button id="openPopupBtn">Open Popup</button>

<!-- JavaScript -->
<script>
  const openPopupBtn = document.getElementById('openPopupBtn');

  openPopupBtn.addEventListener('click', () => {
    const popupWidth = 400;
    const popupHeight = 300;
    const popupFeatures = `width=${popupWidth},height=${popupHeight},left=100,top=100`;

    const popup = window.open('popup-content.html',
     'myPopup', popupFeatures);

    if (popup) {
      // Popup was successfully opened
      console.log('Popup opened!');
    } else {
      // Popup was blocked by the browser
      console.log('Popup was blocked!');
    }
  });
</script>
```

The dialog tag intends to solve all these issues and more. The same intent of **Listing 3** can be expressed using a dialog tag, as shown in **Listing 4**.

There are a number of things that show up in **Listing 4**. At a high level, you create a <dialog> element with an ID of myDialog. You add a button with the ID openDialogBtn to open the dialog. You then add a button with the ID closeBtn to close the dialog. Then you use the showModal() method to open the dialog as a modal window. And finally, you use the close() method to close the dialog.

Listing 4: Using the dialog tag

```
<!-- HTML -->
<dialog id="myDialog">
  <h2>Dialog Title</h2>
  <p>This is a sample dialog box.</p>
  <button id="closeBtn">Close</button>
</dialog>

<button id="openDialogBtn">Open Dialog</button>

<!-- JavaScript -->
<script>
  const openDialogBtn =
    document.getElementById('openDialogBtn');
  const myDialog = document.getElementById('myDialog');
  const closeBtn = document.getElementById('closeBtn');

  openDialogBtn.addEventListener('click', () => {
    myDialog.showModal();
  });

  closeBtn.addEventListener('click', () => {
    myDialog.close();
  });
</script>
```

Compared to **Listing 3**, there are some clear advantages. This dialog blocks interaction with the rest of the page until it's closed. It's modal. This means that you don't have to implement complex workarounds because you can't make the dialog modal using window.popup. Think back to the ticketing application I was talking about a moment ago. If the user is attempting to book a ticket and the ticket is already sold out, you probably don't want the user to continue booking the ticket in the background and attempt to confuse your system. With the modal dialog, you achieve this very easily.

Additionally, the dialog automatically focuses on the first focusable element when it's opened, keeping the user in the flow of the application. Also, the dialog automatically adds ARIA attributes to make it accessible to screen readers.

The only downside of the dialog tag is that the <dialog> tag is not supported in older browsers, so you may need to add fallback code or use a polyfill for older browsers. At the time of writing this article, the dialog tag is supported in Google Chrome, Mozilla Firefox, Microsoft Edge, Opera, and partially supported in Safari.

In case the dialog tag isn't supported by your intended browser, you can easily use dialog-polyfill available as an npm package, or via CDN. The code change is quite simple, as you can see **Listing 5**, effectively using the dialog shown in **Listing 4**, and using polyfill.

Listing 5: Using the dialog tag with polyfills for unsupported browsers

```
<!-- HTML -->
<dialog id="myDialog">
  <h2>Dialog Title</h2>
  <p>This is a sample dialog box.</p>
  <button id="closeBtn">Close</button>
</dialog>

<button id="openDialogBtn">Open Dialog</button>

<!-- JavaScript -->
<script>
  // Check if the browser supports the <dialog> tag
  if (!('showModal' in HTMLDialogElement.prototype)) {
    // If not, load the polyfill
    document.write('<script src=
      "https://cdn.jsdelivr.net/
         npm/dialog-polyfill@0.4.9/
           dialog-polyfill.js"><\/script>');
  }

  // Wait for the polyfill to load
  document.addEventListener('DOMContentLoaded', () => {
    const openDialogBtn =
      document.getElementById('openDialogBtn');
    const myDialog = document.getElementById('myDialog');
    const closeBtn = document.getElementById('closeBtn');

    openDialogBtn.addEventListener('click', () => {
      myDialog.showModal();
    });

    closeBtn.addEventListener('click', () => {
      myDialog.close();
    });
  });
</script>
```

Improvement to HTML Forms

While we're on the topic of dialogs, let's also delve further into accepting inputs from the users that are done via the form tag. The HTML form tag typically has been the mainstay for accepting user input. Recent HTML5 improvements commonly available across modern browsers have made this task a lot easier. Let's dive into some of these new improvements.

The HTML5 form tag now includes a number of new attributes. These attributes are:

- A placeholder attribute to add a hint to form fields
- A required attribute to specify whether a field is mandatory
- A pattern attribute that lets you define a regular expression for field validation
- The min and max attributes that allow you to specify the minimum and maximum values for number and date fields
- A step attribute that lets you define the increment/decrement step for number fields

Let's understand one of these. I pick the min and max attributes. Let's say that I wish to create an HTML form, and I want to ask the user a value for "quantity." I also want to limit this to a minimum value of 1 and a maximum value of 10, with an initial value of 1. This can be achieved using the code snippet below:

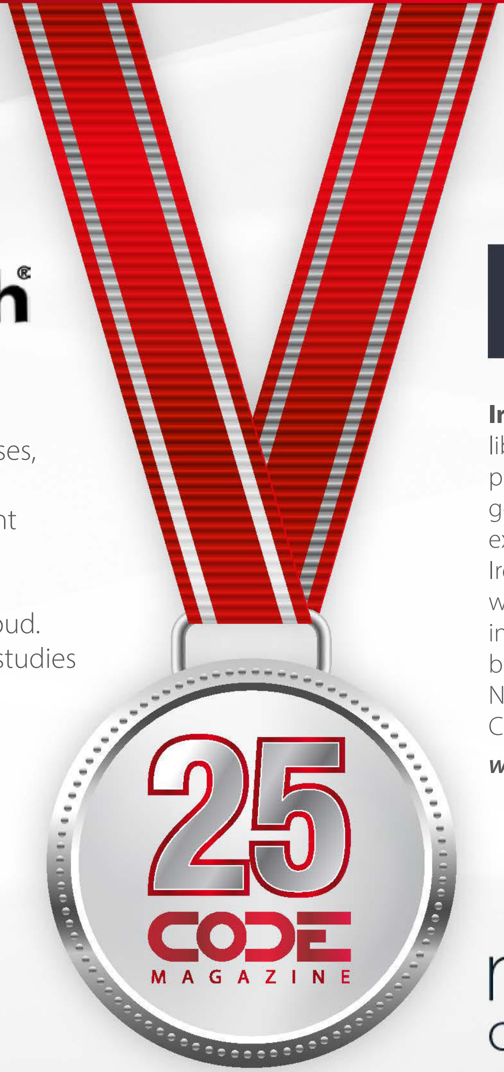
```
<input type="number"
    id="quantity" name="quantity"
     min="1" max="10" value="1">
```

You need to consider older browsers that understand these attributes. This is a job for polyfills. You can find the full code in **Listing 6**. At a high level, you can assume that the HTML form includes a type="number" input field with min and max attributes set to 1 and 10, respectively. Modernizr detects whether the browser supports HTML5 form features, including the min and max attributes. If the browser doesn't support these features, the jQuery polyfill kicks in. The polyfill listens for input events on the #quantity field and checks the value against the min and max attributes. If the value is outside the allowed range, the polyfill adjusts the value to the nearest limit (either min or max).

Speaking of Modernizr, it's an incredibly useful library that you should know about. Modernizr is a JavaScript

25 Years of Publishing

Developers Helping Developers



dtSearch®

dtSearch® - INSTANTLY SEARCH TERABYTES in files, emails, databases, web data. Includes dtSearch's own document filters. Over 25 concurrent search options. Cross-platform APIs C++, Java and current .NET for Win/Linux/macOS. On-premises or in cloud. Visit dtSearch for hundreds of case studies and reviews plus full evaluations.

www.dtsearch.com

IRONSOFTWARE

Iron Software offers a complete suite of C# libraries that streamline document and data processing in .NET applications. Whether you're generating pixel-perfect reports with IronPDF, extracting text from scanned files using IronOCR, or adding barcode functionality with IronBarcode, these tools drop straight into your workflow — so you can focus on building, not battling your tools. No hacks. No workarounds. Just clean, native-feeling C# code that helps you ship faster and smarter.

www.ironsoftware.com

CODE STAFFING

At **CODE Staffing**, we provide flexible software solutions that offer control without compromise and a unique Talent as a Service model that delivers top-tier talent without the overhead of traditional hiring. Our holistic approach, customized for each client, allows companies to scale quickly, access specialized expertise and training, and ensures long-term success—all while maintaining agility.

www.codestaffing.com

next GenAI CONFERENCE

October 7-9, 2025 |
Loews Sapphire Falls Resort, Orlando
Join software developers and business leaders at the second annual **Next GenAI Conference**. Learn to select optimal AI tech stacks, secure organizational buy-in, and maximize ROI on AI investments. Navigate data security, regulatory compliance, and change management in the evolving AI landscape.

www.nextgenaiconf.com

We thank these sponsors who have helped us fullfill our goal of empowering LOB developers.
Their support is the foundation of our silver anniversary.

Listing 10: Using the setCustomValidity method

```
const emailInput = document.getElementById('email');
const emailError = document.getElementById('emailError');

emailInput.addEventListener('input', () => {
  if (emailInput.validity.typeMismatch) {
    emailInput.setCustomValidity(
       'Please enter a valid email address.');
    emailError.textContent =
       'Please enter a valid email address.';
  } else {
    emailInput.setCustomValidity('');
    emailError.textContent = '';
  }
});
```

Listing 6: Using min/max attributes via modernizr and jquery polyfills

```
// Modernizr detection
if (!Modernizr.inputtypes.number ||
       !Modernizr.formvalidation) {
  // jQuery polyfill
  $(document).ready(function() {
    $('#quantity').on('input', function() {
      var min = parseInt($(this).attr('min'));
      var max = parseInt($(this).attr('max'));
      var val = parseInt($(this).val());

      if (val < min) {
        $(this).val(min);
      } else if (val > max) {
        $(this).val(max);
      }
    });
  });
}
```

Listing 7: The output element

```
<form oninput=
  "result.value = parseInt(a.value) + parseInt(b.value)">
  <input type="number" id="a" value="0">
  <input type="number" id="b" value="0">
  <output for="a b" id="result" name="result">0</output>
</form>
```

Listing 8: The progress bar styling customizability

```
progress {
  width: 100%;
  height: 20px;
  border: none;
  border-radius: 10px;
  background-color: #f2f2f2;
}

progress::-webkit-progress-bar {
  background-color: #f2f2f2;
}

progress::-webkit-progress-value {
  background-color: #4CAF50;
}

progress::-moz-progress-bar {
  background-color: #4CAF50;
}
```

Listing 9: Simple email input

```
<form id="emailForm">
    <label for="email">Email:</label>
    <input type="email" id="email" name="email" required>
    <button type="submit">Submit</button>
    <div id="emailError"></div>
</form>
```

library that detects HTML5 and CSS3 features in a user's browser, allowing you to provide fallbacks or polyfills for older browsers that lack support for these features. It will detect support for features, and, based on the detection results, you can provide fallbacks or polyfills to ensure that your web application works across different browsers.

But I digress. Let's get back to HTML5 forms. Another great improvement is the introduction of new HTML5 form elements. Specifically, the datalist element that provides a list of suggested values for a form field, the output element that displays the result of a calculation or user interaction, and the progress element that shows the progress of a task.

Let's look at one of these in action, although I encourage you to explore each of these. **Listing 7** shows the output element in action. **Listing 7** has a simple form with two number input fields (a and b) and an <output> element (result). The oninput attribute specifies a JavaScript expression to be executed when the user interacts with the form (e.g., changes the value of a or b). The JavaScript expression calculates the sum of the values of a and b and assigns it to the value property of the result output element. The for attribute of the <output> element specifies the IDs of the elements that contribute to the calculation (in this case, a and b).

To offer polyfill for this, you're effectively going to have the same logic again. I'm going to skip that because in today's world, every browser you probably care for supports the output tag. Let me leave you with an example of using the progress element. Also, try it out and see what it does.

```
<progress id="progressBar"
   value="50" max="100">
</progress>
```

The cool thing about the progress element is that it's fully scriptable via JavaScript. You just modify the .value property of the element. You can also highly stylize it. **Listing 8** shows a fully stylized progress bar. I'm quite tempted to leave a screenshot of the progress bar here, but try it out.

Speaking of styling and forms, HTML5 also has some enhanced validation and feedback for forms. You could create field types like email and URL and you get automatic validation, but you can now style them with :valid and :invalid css pseudo classes. Or you can use the setCustomValidity() method to enable custom error messages for form fields.

Let's construct a simple example. First, create your user interface, as can be seen in **Listing 9**. This is a simple form that accepts a text box input and specifies its type to be email. Additionally, you're requiring this field by using the required attribute. I also have a div showing any error messages with the validation by the name of "emailError".

Now let's style it. I'm going to keep it simple and specify two classes for valid and invalid. For the CSS gurus among you, you know you can add some very nice animations and effects here. But my code is rather simple. I simply change the border to green or red for valid or invalid, as can be seen below:

```
input:valid {
    border: 2px solid #4CAF50;
}
```

```
input:invalid {
    border: 2px solid #f44336;
}
```

This alone provides the user with some visual cues. But sometimes you want to have custom validation logic and show a custom error message. This is where our good old friend JavaScript comes in, and you leverage the setCustomValidity method, as can be seen in **Listing 10**.

Now let's run this example. When you load the page, it should show you an email field that is highlighted in red, because it's invalid. This can be seen in **Figure 1**.

As I start entering an email, it validates it for me and shows me an appropriate error message, as can be seen in **Figure 2**.

And finally, when I enter a proper email address, the field turns green, as per the CSS styling. This can be seen in **Figure 3**.

Vector Graphics

When HTML was invented, it had some very basic tags that allowed you to do some basic formatting like bold, italic, underline, etc. This was a while ago, when graphics were a luxury. As computers progressed, we were amazed to see a computer screen render an image. And it took forever to download it too. To support this, HTML introduced the IMG tag. As time went on, we came up with newer and better formats for representing images. The first format I remember was BMP, or bitmap for short. There may have been other formats, but at least this is the first format I remember. What I remember about it is that it was incredibly inefficient. The file sizes were huge. My jaw dropped when I saw a JPEG image for the first time. The file size was a fraction of what BMP was, but the quality was nearly the same. Then came GIF, which could even do animations. GIF even allowed me to do transparent backgrounds. Then came PNG, which combined the best of both JPEG and GIF formats. Recently, Apple introduced the HEIC format which offers even better image quality and better compression than JPEG.

All of these formats had one shortcoming. These are all examples of raster graphics. Raster graphics, also known as bitmap graphics, are digital images composed of a grid of tiny squares called pixels. Each pixel has a specific color value, which, when combined, forms the final image. All of these formats are compression algorithms around raster graphics. Because they're pixel based, they're resolution dependent, and the final quality of the image depends on the screen and DPI being used to view an image. Also, the file sizes can get incredibly large as the images get larger. I use a 65" 8K screen as my monitor. Yes, I'm spoiled, but literally every image on the internet looks like crap.

SVGs, or scalable vector graphics, intend to solve this problem. SVG is an XML-based vector image format that allows the creation of dynamic, scalable, and interactive graphics. SVGs are made up of vectors, which are mathematical equations that draw shapes and lines rather than pixels. They can scale up or down without losing resolution. Additionally, their file sizes are tiny in comparison and they are editable in a text editor or via JavaScript.

Listing 11 shows a simple example using the SVG tag available in HTML5. This should produce an image like that shown in **Figure 4**.

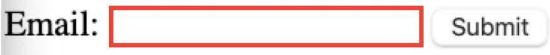
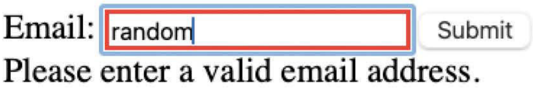


Figure 1: An invalid email with built in validation



Please enter a valid email address.

Figure 2: Email validation rocks

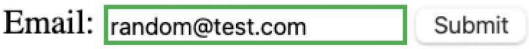


Figure 3: A valid email address, with validation

What's unique about this image is that not only did you express it in plain text, but no matter how much you zoom in or out, this image doesn't lose its resolution.

This opens up some very interesting capabilities. Think about it. This SVG tag that you expressed was in plain text. Theoretically speaking, you could use plain JavaScript code to manipulate this image on the fly. Indeed, SVG supports that, and interesting concepts like animations, and all sorts of visual effects. There is a rather interesting library called D3.js that takes this to the ultimate lengths.

Let's look at a simple example of using D3.js to render out a chart.

Typically, you're given data that looks like **Listing 12**, and the task given is: "turn this into a bar chart." Historically speaking, I would have rendered this as an image on the server side and downloaded that PNG. Very clever! But with SVG, you can do this entirely client side. Here's how.

First, let's define the HTML, as can be seen in **Listing 13**. The key parts here are that I'm referencing D3.js, I've defined a div on the page where my chart will go, and I'm externalizing all my logic in script.js.

Now let's focus on script.js. The first thing I'd like to do is give my chart some dimensions and add an SVG tag to my HTML page to hold my chart. This can be seen in **Listing 14**.

SPONSORED SIDEBAR:

AI Executive Briefing

Experience the game-changing impact of **AI** through **CODE Consulting's** Executive Briefing service. Uncover the immense potential and wide-ranging benefits of **AI** in every industry. Our briefing provides **strategic guidance** for seamless implementation, covering crucial aspects such as infrastructure, talent acquisition, and leadership.

Discover how to effectively **integrate AI** and propel your organization into future success.

Contact us today to schedule your executive briefing and embark on a journey of AI-powered growth. www.codemag.com/ai

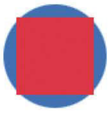


Figure 4: SVG output

Listing 11: SVG Graphics

```
<svg width="100" height="100">
  <circle cx="50" cy="50" r="40" fill="#007bff" />
  <rect x="20" y="20" width="60"
     height="60" fill="#dc3545" />
</svg>
```

Listing 12: Some random data

```
const data = [
    { label: 'Jan', value: 100 },
    { label: 'Feb', value: 150 },
    { label: 'Mar', value: 200 },
    { label: 'Apr', value: 120 },
    { label: 'May', value: 180 },
    { label: 'Jun', value: 130 },
    { label: 'Jul', value: 80 },
    { label: 'Aug', value: 20 },
    { label: 'Sep', value: 170 }
];
```

Listing 13: My chart HTML file

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport"
       content="width=device-width, initial-scale=1.0">
    <script src="https://d3js.org/d3.v7.min.js"></script>
</head>
<body>
    <div id="mychart"></div>
    <script src="script.js"></script>
</body>
</html>
```

Listing 14: Add my svg to the page

```
// set the dimensions and margins of the graph
const margin =
   { top: 10, right: 30, bottom: 90, left: 40 },
    width = 600 - margin.left - margin.right,
    height = 400 - margin.top - margin.bottom;

// append the svg object to the body of the page
const svg = d3.select("#mychart")
    .append("svg")
    .attr("width", width + margin.left + margin.right)
    .attr("height", height + margin.top + margin.bottom)
    .append("g")
    .attr("transform",
       `translate(${margin.left},${margin.top})`);
```

Listing 15: Setup the X axis

```
const x = d3.scaleBand()
    .range([0, width])
    .domain(data.map(d => d.label))
    .padding(0.2);
svg.append("g")
    .attr("transform", `translate(0,${height})`)
    .call(d3.axisBottom(x))
    .selectAll("text")
    .attr("transform", "translate(-10,0)rotate(-45)")
    .style("text-anchor", "end");
```

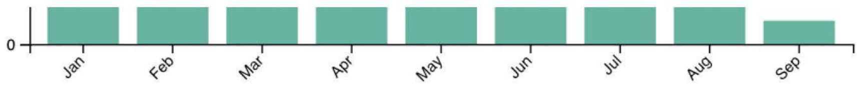


Figure 5: The X-axis

Listing 16: Setup the Y axis

```
const y = d3.scaleLinear()
    .domain([0, 200])
    .range([height, 0]);
svg.append("g")
    .call(d3.axisLeft(y));
```

Listing 17: Plot the bars

```
svg.selectAll("mybar")
    .data(data)
    .join("rect")
    .attr("x", d => x(d.label))
    .attr("width", x.bandwidth())
    .attr("fill", "#69b3a2")
    .attr("height", d => height - y(0))
    .attr("y", d => y(0))
```

Let's set up the X axis, which can be seen in **Listing 15**. This code creates a band scale using D3.js, which is used to map the labels to a continuous range of values to the X axis. The .range([0, width]) specifies the range of the scale, which is the width of the SVG element. You'll see that .domain(data.map(d => d.label)) specifies the domain of the scale, which is the list of labels from the data, and .padding(0.2) adds a padding of 20% between each band, making the bars more spaced out.

Once this is set up, the svg.append line creates a new group element (g) in the SVG and appends it to the SVG element. It then sets the transformation attribute of the group to translate it to the bottom of the SVG (by setting the Y-coordinate to the height of the SVG). Then you call the d3.axisBottom(x) function to create a bottom-oriented axis using the previously defined X scale. Finally, using the transform attribute, you rotate the text slightly so it looks nice in the final generated chart, which can be seen in **Figure 5**.

Now let's focus on the Y axis. The Y-axis is slightly more interesting. You have a range of values to plot, so you probably want to scan the values and find the min and max and scale your axis accordingly. That is just rote math, so to keep my code simple, I'll just go with a max of 200 and hardcode it in my code. The full code for setting up the Y axis can be seen in **Listing 16**.

Next, you need to plot the bars. The code to plot the bars can be seen in **Listing 17**. Let's dissect this code bit by bit.

The svg.selectAll command selects all elements with the tag name "mybar" within the SVG element, of which, initially, there are none because you haven't plotted anything yet. It then binds the data array to these elements using the data() method.

```
svg.selectAll("mybar")
    .data(data)
```

You use the join method to create, update, or remove elements based on the bound data. For example, in the code, you're creating a new rect element based on the data array.

```
.join("rect")
```

Then, you set the label, width (bandwidth), and fill color of the rectangle, as shown below:

```
.attr("x", d => x(d.label))
.attr("width", x.bandwidth())
.attr("fill", "#69b3a2")
```

Finally, you set the initial height and y position of each rectangle.

```
.attr("height", d => height - y(0))
.attr("y", d => y(0))
```

But why are the initial positions zero?

It's because you're going to add animations. You want the bars to get plotted from left to right and grow from bottom to top. The code for that can be seen in **Listing 18**. The first three lines of this code basically describe

Listing 18: Add some animation for fun

```
svg.selectAll("rect")
    .transition()
    .duration(800)
    .attr("y", d => y(d.value))
    .attr("height", d => height - y(d.value))
    .delay((d, i) => { console.log(i); return i * 100 })
```

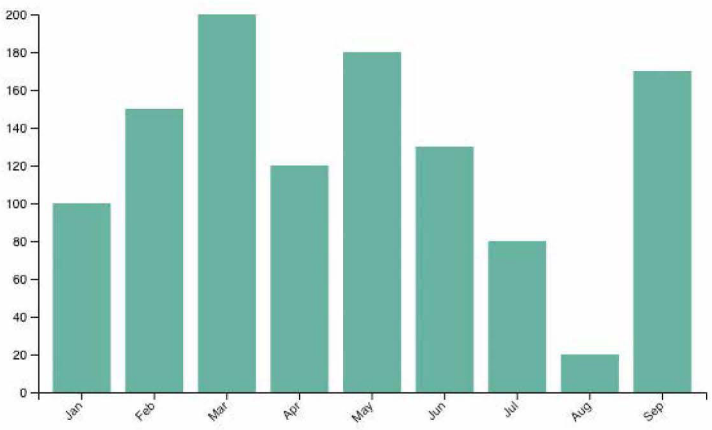


Figure 6: The fancy chart

that you wish to have a transition of duration 800 milliseconds. And the last three lines specify the values you wish to change in this transition.

All put together, the chart looks like **Figure 6**.

A picture doesn't do animations justice, so I put this code on a CodePen here: https://codepen.io/maliksahil/pen/gbOmQZd.

Summary

What I love the most about being a software developer is that the constant pace of change never ceases to amaze me. Imagine if I'd decided to become a doctor or a pilot. From the moment I started my career to now, the basic concepts would have remained the same. As a doctor, I'll just tell my patients to exercise more, and if all else fails, blame it on age or insurance. As a pilot, it's basically the same controls, rules, flight controls, and concepts. Imagine what a pilot would feel like if they'd gone through the same transition as we did in the field of programming. Just 20 years ago, we used to use dial up internet, waiting patiently for an image to download, which could take 30 minutes. And boy did it suck when the phone rang in the middle of that download and all you could see was the top half. Then came JPG, and the whole image downloaded within a minute. I was simply blown away. Then we saw charts, rendered as images on the server, and we could create impressive dashboards delivered via the web. And now, you have fancy animations running entirely client-side on your phone.

Mind blown, simply mind blown. This pace isn't going to slow down. And I'm so glad I picked this field of work. There's never a boring day.

Until next time, happy coding!

Sahil Malik

dtSearch®

Instantly Search Terabytes

Enterprise and developer products have:

- over 25 different search features
- credit card search and other forensics-oriented options
- efficient multithreaded indexing and searching

dtSearch's document filters support:

- local and remote "Office" files, PDFs, compression formats, etc.
- emails with multilevel attachments
- a wide variety of databases
- web data

Developers:

- SDKs for Win / Linux / macOS
- APIs for current .NET / C++ / Java
- faceted search, granular data classification and other API options
- deploy on-premises or in cloud

Visit dtSearch.com

- **for hundreds of reviews and case studies**
- **for fully-functional enterprise and developer evaluations**

The Smart Choice for Text Retrieval® since 1991

dtSearch.com | 1-800-IT-FINDS

ONLINE QUICK ID 2505031

Exploring .NET MAUI: Popups, Messages, and Data Validation

Through this article series, you've created several .NET MAUI pages, performed navigation, used data binding, and worked with the MVVM and DI design patterns. As you created your view models, you've set information and exception message properties. In this article, you'll build reusable components to display information, error, and validation messages on your pages. To validate user input, you're going to use data annotation attributes such as [Required] and [Range]. With just a little generic code, you can add validation to your .NET MAUI applications and display the error messages from these data annotation attributes. Sometimes you require a pop-up dialog to ask the user a question, get a little piece of data, or maybe just provide some information to the user. There are a few different dialogs you can use in .NET MAUI and you'll start this article by exploring these.

Paul D. Sheriff
http://www.pdsa.com

Paul has been in the IT industry since 1984. In that time he has successfully assisted hundreds of companies architect software applications to solve their toughest business problems. Paul has been a teacher and mentor through various mediums such as video courses, blogs, articles and speaking engagements at user groups and conferences around the world. Paul has multiple courses in the www.pluralsight.com (https://app.pluralsight.com/profile/author/paul-sheriff) library on topics ranging from C#, LINQ, JavaScript, Angular, MVC, WPF, XML, jQuery and Bootstrap. Check out his videos and courses on YouTube (https://www.youtube.com/%40pauldsheriff) as well.

Display Your Own Page as a Modal Dialog

Any page you create may be used as a modal dialog by calling the PushModalAsync() method on the **Navigation** object. For this application, before a user can log in, they must read and accept a privacy policy, as shown in **Figure 1**. This policy page is displayed as a modal dialog. After the user accepts the terms, the Login button becomes enabled so the user can continue using the application. If they don't accept the policies, they're never allowed to log in. To build this functionality, create a privacy policy page and the corresponding view model, as well as a log in view model, and an application settings class to hold the Boolean of whether or not the user accepted the policies.

Create the Privacy Policy View Model

You're not going to add all the functionality for enabling or disabling the Login button yet. First, let's learn how to display a modal dialog and return to the calling page by clicking on a button. Right mouse-click on the **MauiViewModelClasses** folder and add a new class named **PrivacyPolicyViewModel**. Replace the entire contents of this new file with the code shown in **Listing 1**. At this point, you don't need a corresponding view model class in the ViewModelLayer project as you are just creating the functionality to call pages in the front-end application.

This view model contains two Command objects; AcceptCommand and DontAcceptCommand, which will be mapped to the two buttons you saw on the privacy policy page in **Figure 1: Before a user can log in, they must click the I ACCEPT button on the privacy policy page.**. To dismiss a modal dialog call the PopModalAsync() method, as shown in the two methods created in this view model; AcceptPolicyAsync() and DontAcceptPolicyAsync().

Create the Privacy Policy Page

Right mouse-click on the Views folder and select **Add > New Item…** from the menu. Click on the **.NET MAUI** tab, select the **.NET MAUI ContentPage (XAML)** template, set the **Name** to **PrivacyPolicyView**, and click on the Add button. Set the **Title** attribute to **Privacy Policy**. Add an XML namespace to reference the partial views in this project, and another to reference the view model you just created.

```
xmlns:partial="clr-namespace:
  AdventureWorks.MAUI.ViewsPartial"
xmlns:vm="clr-namespace:
  AdventureWorks.MAUI.MauiViewModelClasses"
```

Add the **x:DataType** attribute to the ContentPage starting tag so you can connect the buttons to the commands created in the view model class.

```
x:DataType="vm:PrivacyPolicyViewModel"
```

Replace the <VerticalStackLayout> element with the XAML shown in **Listing 2**. Due to space limitations in this publication, I did not include all the text shown in **Figure 2**. Feel free to add additional text if you wish to mimic **Figure 2**.

Open the **Views\PrivacyPolicy.xaml.cs** file and replace the entire contents of this file with the code shown in **Listing 3**. This code should look familiar as it's typical of most pages you have developed in this article series. The PrivacyPolicyViewModel is injected into the constructor and is assigned to the **_ViewModel** variable. In the OnAppearing() method, you set the **BindingContext** of the page to the instance of the view model injected.

To be injected into the privacy policy page, the **PrivacyPolicyViewModel** needs to be added to the DI container. Open the **ExtensionClasses\ServiceExtensions.cs** file and add the following line of code to the AddViewModelClasses() method:

```
services.AddScoped<MauiViewModelClasses.
  PrivacyPolicyViewModel>();
```

The privacy policy page also needs to be a part of the DI process, so add the page to the AddViewClasses() method in this same class, as shown in the following code:

```
services.AddScoped<PrivacyPolicyView>();
```

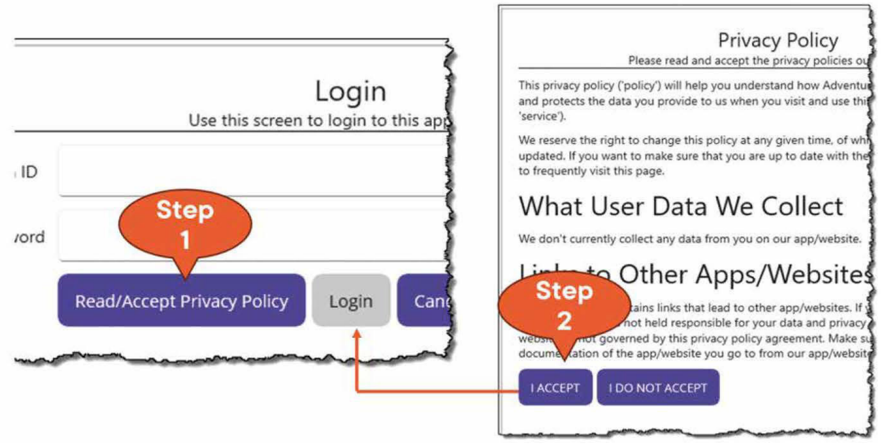
Create the Login View Model

The log in page needs a view model to hold the log in ID and password, and to call the privacy policy page. Right mouse-click on the **MauiViewModelClasses** folder and add a new class named **LoginViewModel**. Replace the entire contents of this new file with the code shown in **Listing 4**. At this point, you don't need a corresponding view model class in the ViewModelLayer project as you're just creating the functionality to call pages in the front-end application.

The code in this front-end view model follows the same design pattern you've been using with the other view models created in this article series. However, the log in view model is going to accept an instance of the **PrivacyPolicyViewModel** as it's needed to create a modal instance of the privacy policy page. Create a Command property named **PrivacyPolicyDisplay** from which you can call the method PrivacyPolicyDisplayAsync() to display the privacy policy page. The page is displayed as a modal dialog by passing a new instance of the privacy policy page to the PushModalAsync() method of the **Navigation** object.

You must now include the log in view model and the log in page in dependency injection. Open the **ExtensionClasses\ServiceExtensions.cs** file and add the following line of code to the AddViewModelClasses() method:

```
services.AddScoped<MauiViewModelClasses.
  LoginViewModel>();
```


Login
Use this screen to login to this ap
ID
ord
Step 1
Read/Accept Privacy Policy
Login
Can
Privacy Policy
Please read and accept the privacy policies o
Step 2
I ACCEPT
I DO NOT ACCEPT

This privacy policy ('policy') will help you understand how Adventu and protects the data you provide to us when you visit and use thi 'service').

We reserve the right to change this policy at any given time, of wh updated. If you want to make sure that you are up to date with the to frequently visit this page.

What User Data We Collect

We don't currently collect any data from you on our app/website.

Other Apps/Websites

tains links that lead to other app/websites. If not held responsible for your data and privacy governed by this privacy policy agreement. Make s ation of the app/website you go to from our app/websit

Figure 1: Before a user can log in, they must click the I ACCEPT button on the privacy policy page.

The log in page also needs to be a part of the DI process, so add the page to the AddViewClasses() method in this same class, as shown in the following code:

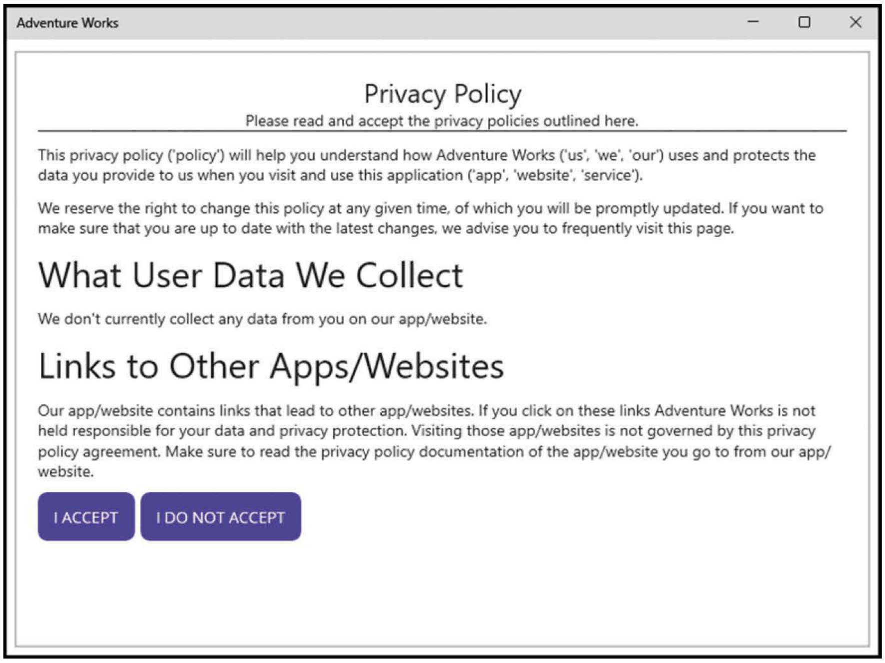
```
services.AddScoped<LoginView>();
```

Listing 1: Create a privacy policy view model class to execute commands.

```
using Common.Library;
using System.Windows.Input;

namespace AdventureWorks.MAUI
  .MauiViewModelClasses;

public class PrivacyPolicyViewModel
  : ViewModelBase {
  #region Commands
  public ICommand? AccceptCommand
    { get; private set; }
  public ICommand? DontAcceptCommand
    { get; private set; }
  #endregion

  #region Init Method
  public override void Init() {
    base.Init();

    // Create commands for this view
    AccceptCommand = new Command(
      async () => await AcceptPolicyAsync());
    DontAcceptCommand = new Command(
      async () => await DontAcceptPolicyAsync());
  }
  #endregion

  #region AcceptPolicyAsync Method
  public async Task AcceptPolicyAsync() {
    await Shell.Current.Navigation
      .PopModalAsync();
  }
  #endregion

  #region DontAcceptPolicyAsync Method
  protected async Task DontAcceptPolicyAsync() {
    await Shell.Current.Navigation
      .PopModalAsync();
  }
  #endregion
}
```

Listing 2: Add a normal content page and add some text to display a privacy policy.

```
<Border Style="{StaticResource Border.Page}">
  <ScrollView>
    <Grid Style="{StaticResource Grid.Page}"
          RowDefinitions="Auto,Auto,Auto,Auto,
                          Auto,Auto,Auto,Auto">

      <partial:HeaderView
        ViewTitle="Privacy Policy"
        ViewDescription="Please read and accept
         the privacy policies outlined here." />

      <Label Grid.Row="1"
        Text="This privacy policy..." />

      <Label Grid.Row="2"
        Text="We reserve the right..." />

      <Label Grid.Row="3"
        FontSize="Large"
        Text="What User Data We Collect" />

      <Label Grid.Row="4"
        Text="We don't currently collect
        any data from you on our app/website." />

      <Label Grid.Row="5"
        FontSize="Large"
        Text="Links to Other Apps/Websites" />

      <Label Grid.Row="6"
             Text="Our app/website contains..." />

      <HorizontalStackLayout Grid.Row="7">
        <Button Text="I ACCEPT"
          Command="{Binding AccceptCommand}" />
        <Button Text="I DO NOT ACCEPT"
          Command="{Binding DontAcceptCommand}" />
      </HorizontalStackLayout>

    </Grid>
  </ScrollView>
</Border>
```



Privacy Policy

Please read and accept the privacy policies outlined here.

This privacy policy ('policy') will help you understand how Adventure Works ('us', 'we', 'our') uses and protects the data you provide to us when you visit and use this application ('app', 'website', 'service').

We reserve the right to change this policy at any given time, of which you will be promptly updated. If you want to make sure that you are up to date with the latest changes, we advise you to frequently visit this page.

What User Data We Collect

We don't currently collect any data from you on our app/website.

Links to Other Apps/Websites

Our app/website contains links that lead to other app/websites. If you click on these links Adventure Works is not held responsible for your data and privacy protection. Visiting those app/websites is not governed by this privacy policy agreement. Make sure to read the privacy policy documentation of the app/website you go to from our app/website.

Figure 2: You can call any of your own pages as a modal dialog.

Listing 3: Set the privacy policy view model into the BindingContext of the page.

```
using AdventureWorks.MAUI.MauiViewModelClasses;

namespace AdventureWorks.MAUI.Views;

public partial class PrivacyPolicyView
  : ContentPage {
  public PrivacyPolicyView(
    PrivacyPolicyViewModel viewModel) {
    InitializeComponent();

    _ViewModel = viewModel;
  }

  private readonly PrivacyPolicyViewModel
    _ViewModel;

  protected override void OnAppearing() {
    base.OnAppearing();

    BindingContext = _ViewModel;
  }
}
```

Modify the Login View Page

In the second article of this series (https://codemag.com/Article/2409041/Exploring-.NET-MAUI-Styles-Navigation-and-Reusable-UI), you created the UI for the log in page, but didn't really hook up the view model. Let's do that now by opening the **Views\LoginView.xaml** file and adding an XML namespace to reference the login view model class.

```
xmlns:vm="clr-namespace:
  AdventureWorks.MAUI.MauiViewModelClasses"
```

Add an **x:DataType** attribute to the ContentPage starting tag so you can reference any data and commands in the log in view model class.

```
x:DataType="vm:LoginViewModel"
```

You're going to add a third button to this log in page, as shown in **Figure 1**, because, on a mobile device, this can cause the buttons to be cropped on the right side of the screen. Replace the entire <HorizontalStackLayout> that contains the Login and Cancel buttons with a FlexLayout, as shown in the code below.

```
<FlexLayout Grid.Row="3"
            Grid.Column="1"
            Wrap="Wrap"
            Direction="Row">
  <Button Text="Read/Accept Privacy Policy"
    Style="{StaticResource flexButton}"
    Command="{Binding PrivacyPolicyDisplay}" />
  <Button Text="Login"
    Style="{StaticResource flexButton}"
    Margin="5" />
  <Button Text="Cancel"
    Style="{StaticResource flexButton}"
    Margin="5" />
</FlexLayout>
```

To ensure the buttons within the FlexLayout control have a little separation in case they should wrap to another line, add a <ContentPage.Resources> element just after the <ContentPage> starting tag. Add a margin to each button with a spacing of five, as shown in the code below:

```
<ContentPage.Resources>
  <Style TargetType="Button"
```

Listing 4: Create a view model for the login page.

```
using Common.Library;
using System.Windows.Input;

namespace AdventureWorks.MAUI.MauiViewModelClasses;

public class LoginViewModel : ViewModelBase {
  public LoginViewModel(
    PrivacyPolicyViewModel pviewModel) {
    _PrivacyViewModel = pviewModel;
  }

  #region Private Variables
  private readonly PrivacyPolicyViewModel
    _PrivacyViewModel;
  #endregion

  #region Commands
  public ICommand? PrivacyPolicyDisplay
    { get; private set; }
  #endregion
```

```

  #region Init Method
  public override void Init() {
    base.Init();

    // Create commands for this view
    PrivacyPolicyDisplay = new Command(
      async () => await
        PrivacyPolicyDisplayAsync());
  }
  #endregion

  #region PrivacyPolicyDisplayAsync Method
  public async Task PrivacyPolicyDisplayAsync() {
    await Shell.Current.Navigation
      .PushModalAsync(new Views
      .PrivacyPolicyView(_PrivacyViewModel));
  }
  #endregion
}
```

```
         x:Key="flexButton">
    <Setter Property="Margin"
            Value="5" />
  </Style>
</ContentPage.Resources>
```

The log in page needs to access the log in view model to be able to call the PrivacyPolicyDisplay command. Open the **Views\LoginView.xaml.cs** file and replace the entire contents of this file with the code shown in **Listing 5**. This code injects the **LoginViewModel** into the constructor and assigns it to the private variable named **_ViewModel**.

Try It Out

Run the application and click on the **Login** menu. Click on the **Read Privacy Policy** button to display the privacy policy page shown in **Figure 2**. Click on either of the buttons to return to the log in page.

Note in **Figure 2** that the menus and all other navigation is removed from the shell. No other keys or clicking anywhere else removes this modal dialog. You must click on one of the buttons presented.

Returning Value(s) from a Modal Page

To determine which button was pressed when the modal page was closed, you need some way to communicate that information to the calling page. You can do this using the [QueryProperty] attribute and modifying the OnAppearing() event procedure. Open the **Views\LoginView.xaml.cs** file and add two public properties.

```
public string? LastPage { get; set; }
public bool IsPrivacyPolicyAccepted { get; set; }
```

Add two [QueryProperty] attributes to map the values passed when navigating to each property.

```
[QueryProperty(nameof(LastPage), "lastPage")]
[QueryProperty(nameof(IsPrivacyPolicyAccepted),
  "isPrivacyPolicyAccepted")]
```

In the OnAppearing() event procedure, shown in **Listing 6**, check to ensure that the **LastPage** property is not null. If it isn't, check to see if the **LastPage** is set to the value **PrivacyPolicyView**. If both conditions are true, you can check the **IsPrivacyPolicyAccepted** property to determine which button was pressed.

Before you use the PopModalAsync() method in the privacy view model to return to the login page after clicking on a button. Let's change that to use the GoToAsync() method passing in "..", which is the same as popping a page off the navigation stack. Onto the "..", add two query parameters: **lastPage** and **isPrivacyPolicyAccepted**. Open the **MauiViewModelClasses\PrivacyPolicyViewModel.cs** file and locate the AcceptPolicyAsync() method and modify it as shown in the following code:

```
public async Task AcceptPolicyAsync() {
  string ret = "..?lastPage=";
  ret += $"{nameof(Views.PrivacyPolicyView)}";
  ret += "&isPrivacyPolicyAccepted=true";

  await Shell.Current.GoToAsync(ret);
}
```

Listing 5: Add the view model to the login page.

```
using AdventureWorks.MAUI.MauiViewModelClasses;

namespace AdventureWorks.MAUI.Views;

public partial class LoginView : ContentPage
{
  public LoginView(LoginViewModel viewModel) {
    InitializeComponent();

    _ViewModel = viewModel;
  }

  private readonly LoginViewModel _ViewModel;

  protected override void OnAppearing() {
    base.OnAppearing();

    BindingContext = _ViewModel;
  }
}
```

Listing 6: Add code to determine what page called this page.

```
protected override void OnAppearing() {
  base.OnAppearing();

  BindingContext = _ViewModel;

  if (LastPage != null &&
      LastPage ==
        nameof(Views.PrivacyPolicyView)) {

    if (IsPrivacyPolicyAccepted) {
      Console.WriteLine(
        "Privacy Policy was Accepted");
    }
    else {
      Console.WriteLine(
        "Privacy Policy was NOT Accepted");
    }
  }
  LastPage = null;
}
```

You need to do the same procedures in the DontAcceptPolicyAsync() method as shown in the following code. Be sure to pass a false value to the **isPrivacyPolicyAccepted** parameter.

```
protected async Task DontAcceptPolicyAsync() {
  string ret = "..?lastPage=";
  ret += $"{nameof(Views.PrivacyPolicyView)}";
  ret += "&isPrivacyPolicyAccepted=false";

  await Shell.Current.GoToAsync(ret);
}
```

Try It Out

Set a breakpoint on the line of code **if (IsPrivacyPolicyAccepted) in** the OnAppearing() method of the log in page. Run the application and click on the **Login** menu. Click on either button and you should hit the breakpoint in the OnAppearing() event.

Add Application Settings Class

You can now display the privacy page as a modal dialog from the log in page and get the value back from whichever button was pressed. You now need someplace to record that the rest of the application knows whether the terms were accepted. In most applications, you have a global settings class to hold information about the entire application. One of those settings is whether the user has accepted the privacy policies. This property is set when the value is returned from the privacy policy page. Let's create that global settings class. Right mouse-click on the AdventureWorks.MAUI project and add a new folder named **ConfigurationClasses**. Right mouse-click on the ConfigurationClasses folder and add a new class named **AppSettings**. Replace the entire contents of this new file with the code shown in **Listing 7**.

This **AppSettings** class needs to participate in DI so you must register it with the DI container. Open the **ExtensionClasses\ServiceExtensions.cs** file and add a new method named AddOtherClasses(), as shown in the following code. Only a single instance of this class should be available throughout the entire application, so register it with DI using the AddSingleton() method.

```
private static void AddOtherClasses(
  IServiceCollection services) {
  // Add Other Classes
  services.AddSingleton<
    ConfigurationClasses.AppSettings>();
}
```

This new method needs to be called by the public method on the **ServiceExtensions** class. Locate and modify the AddServicesToDIContainer() method and call the AddOtherClasses() method, as shown in the code below:

```
public static void AddServicesToDIContainer(
  this IServiceCollection services) {
  // Add Repository Classes
  AddRepositoryClasses(services);
  // Add View Model Classes
  AddViewModelClasses(services);
  // Add View Classes
  AddViewClasses(services);
  // Add Other Classes
  AddOtherClasses(services);
}
```

Add Settings Class to Privacy Policy View Model

Open the **MauiViewModelClasses\LoginViewModel.cs** file and add a **Using** statement at the top of the file. This namespace is where the **AppSettings** class is located.

```
using AdventureWorks.MAUI.ConfigurationClasses;
```

Create a new private variable to hold the instance of the **AppSettings** class.

```
private AppSettings _Settings;
```

Add a public property to expose the **AppSettings** object so you can bind the **IsPrivacyPolicyAccepted** property to the Login button on the page.

```
#region Public Properties
public AppSettings Settings {
  get { return _Settings; }
  set { _Settings = value; }
}
#endregion
```

Add a constructor to the **LoginViewModel** class and inject an instance of the **AppSettings** class, as shown in the following code snippet:

```
public LoginViewModel(
  PrivacyPolicyViewModel pviewModel,
  AppSettings settings) {
  _PrivacyViewModel = pviewModel;
  _Settings = settings;
}
```

Open the **Views\LoginView.xaml.cs** file and modify the OnAppearing() method to look like the following. Once you determine if the last page was the privacy policy page, set the **Settings.IsPrivacyPolicyAccepted** property to the value returned from the that page.

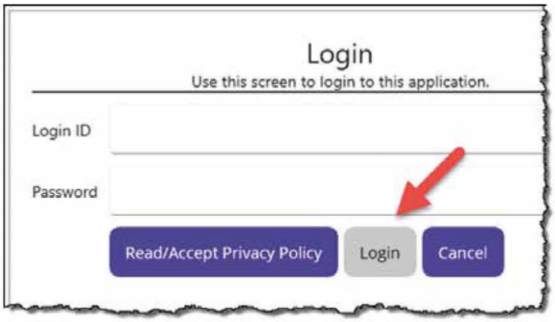
```
protected override void OnAppearing() {
  base.OnAppearing();

  BindingContext = _ViewModel;

  if (LastPage != null &&
      LastPage ==
        nameof(Views.PrivacyPolicyView)) {
    _ViewModel.Settings.IsPrivacyPolicyAccepted
      = IsPrivacyPolicyAccepted;
  }
  LastPage = null;
}
```

Open the **Views\LoginView.xaml** file and locate the Login button. Add the **IsEnabled** attribute to bind to the **Settings.IsPrivacyPolicyAccepted** property in the **LoginViewModel** class.

```
<Button Text="Login"
  IsEnabled="{Binding
    Settings.IsPrivacyPolicyAccepted}" />
```

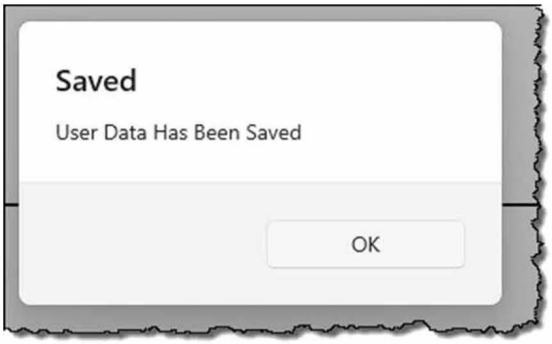
Listing 7: Add a new class

```
using Common.Library;

namespace AdventureWorks.MAUI.ConfigurationClasses;

public class AppSettings : CommonBase {
  #region Private Variables
  private bool _IsPrivacyPolicyAccepted;
  #endregion

  #region Public Properties
  public bool IsPrivacyPolicyAccepted {
    get { return _IsPrivacyPolicyAccepted; }
    set {
      _IsPrivacyPolicyAccepted = value;
      RaisePropertyChanged(
        nameof(IsPrivacyPolicyAccepted));
    }
  }
  #endregion
}
```



Login
Use this screen to login to this application.
Login ID
Password
Read/Accept Privacy Policy
Login
Cancel

Figure 3: The login button is disabled until the privacy policy has been read and accepted.



Saved

User Data Has Been Saved

Figure 4: An alert dialog has a title, a message, and a cancel button.

Try It Out

Run the application and click on the **Login** menu. Notice that the Login button is disabled, as shown in **Figure 3**. Click on the Read/Accept Privacy Policy button and click the I ACCEPT button. Upon returning to the log in page, you should see that the Login button is enabled. Click on the Read/Accept Privacy Policy button one more time, but this time click the I DON'T ACCEPT button. You should see that the Login button is now disabled on the log in page.

Working with Pop-Up Dialogs

Sometimes in your business applications you want to display a simple pop-up message to a user or maybe ask them a Yes/No question. Instead of creating your own page to do this, .NET MAUI supplies you with a few different pop-up dialogs to use. .NET MAUI has three classes for displaying different prompts. These methods, DisplayAlert(), DisplayPromptAsync(), and DisplayActionSheet() are available on the **Page** class. The pop-up rendered will be different based on the operating system you're running your application upon.

The ContentPage inherits from the **Page** class, so these methods are available on all your pages. This means you can call these methods in the code-behind of any page. When called from within the code-behind, the dialogs are displayed on the UI thread. This is important, because if you tried calling these methods from a view model in your ViewModelLayer project, there's no guarantee that you're on the UI thread and thus an exception may be raised. Also, your ViewModelLayer project would no longer be reusable as this would make your view models dependent on .NET MAUI and thus could not be reused.

You may call these dialogs from the view model classes you created within the .NET MAUI application. Because these view models are injected into the code-behind of a ContentPage class they are running on the UI thread. Use the **Shell.Current.CurrentPage** object to invoke each of the popup methods to display these dialogs.

Display an Alert Dialog

The DisplayAlert() method on the ContentPage object might be used to provide confirmation that an action was successful. The user must click on the button of this dialog to continue. The first three parameters you pass to the DisplayAlert() method are a title, a message, and the text to display on the button, as shown in **Figure 4: An alert dialog has a title, a message, and a cancel button.**. The title is displayed in a large, bold font. The message is displayed immediately under the title in a smaller, non-bold font. Open the **MauiViewModelClasses\UserViewModel.cs** file and locate the SaveAsync() method and immediately after the **if(ret != null)** statement, add the following code:

```
await Shell.Current.CurrentPage.DisplayAlert
  ("Saved", "Data Has Been Saved", "OK");
```

Try It Out

Run the application, click on the **Users** menu, then click on the edit button on a specific user. Press the **Save** button and you should see a dialog appear as shown in **Figure 4**. To clear this dialog on Windows, click the **OK** button or press the escape key. On mobile platforms you may click on the OK button, or just press anywhere outside of the button to clear it.

I wouldn't use a dialog in this manner. I don't think you need to provide confirmation when an operation succeeds. I like to only inform a user when something goes wrong. When you eventually write the SaveAsync() method in the **UserViewModel** declared in the ViewModelLayer set the **InfoMessage** property to some text to inform the user that the record was saved successfully. This message will appear briefly in the information message area. This is confirmation enough to the user that their operation was successful. This makes using this dialog unnecessary. Stop the application and remove this line of code to display this dialog.

Alert With Yes/No for Deleting Data

You may also pass four string parameters to the DisplayAlert() method. The fourth parameter becomes the cancel button, and the third parameter becomes the accept button. When the accept button is pressed, **true** is returned from this method. When the cancel button is pressed, **false** is returned from this method.

On the user and product lists, you have a delete button represented by a trash can icon. When a user clicks on that button, before deleting the items, you should ask the user if they really want to perform this operation. Use the same DisplayAlert() method but add two buttons, **Yes** and **No**, to the dialog. Before calling the DisplayAlert() method, let's add the appropriate commanding infrastruc-

Listing 8: Use the DisplayAlert() method to ask the user a yes/no question.

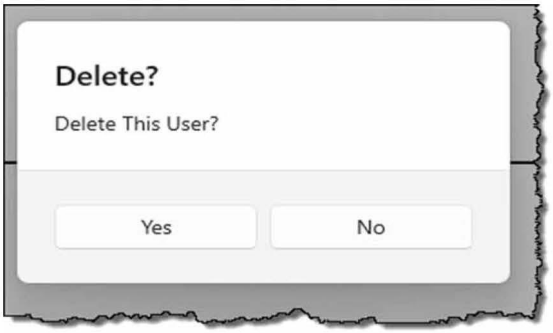
```
#region DeleteAsync Method
public async Task<bool> DeleteAsync() {
  bool ret = false;

  bool perform = await Shell.Current
    .CurrentPage.DisplayAlert(
    "Delete?", "Delete This User?",
    "Yes", "No");
  if (perform) {
    // TODO: Perform delete here
    User? response = new();

    if (response != null) {
      // Redisplay List
      await GetAsync();
    }
  }

  return ret;
}
#endregion
```

ture. Open the **MauiViewModelClasses\UserViewModel.cs** file and add a new ICommand, as shown in the following code:

```
public ICommand? DeleteCommand { get; private set; }
```

In the Init() method create an instance of the Command class and assign it to this **DeleteCommand** property.

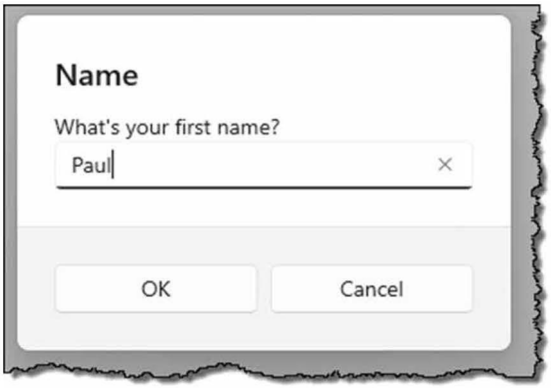
```
DeleteCommand = new Command(
  async () => await DeleteAsync());
```



Delete?

Delete This User?

Figure 5: An alert may have two buttons that return a true or false depending on which one is clicked.



Name

What's your first name?

Paul

Figure 6: A prompt allows you to enter a single piece of information.

Add the DeleteAsync() method referenced by the **DeleteCommand** property, as shown in **Listing 8**. This method declares a variable named **ret** to return a true or false value. The DisplayAlert() method is invoked passing in "Delete?" for the title, and "Delete This User?" for the message. The accept button is displayed with the text "Yes", while the cancel button is displayed with the text "No". If the accept button is clicked a true value is returned and assigned to the **perform** variable. In the **if** statement is call the DeleteAsync() in the **UserViewModel** contained in the ViewModelLayer project. This method returns an instance of the user object that was deleted. If it isn't deleted, a null value is returned. If the item is deleted, invoke the GetAsync() method again to refresh the list with all the values from the data store. This part is optional, you could just delete the user object returned from the list. Feel free to code this either way you wish.

Open the **Views\UserListView.xaml** file and locate the existing delete button within the ListView. Add the **CommandParameter** and **Command** attributes to call the **DeleteCommand** property you added to your view model.

```
<ImageButton Source="trash.png"
  CommandParameter="{Binding UserId}"
  Command="{Binding
    Source={x:Reference UserListPage},
    Path=BindingContext.DeleteCommand}"
  ToolTipProperties.Text="Delete User" />
```

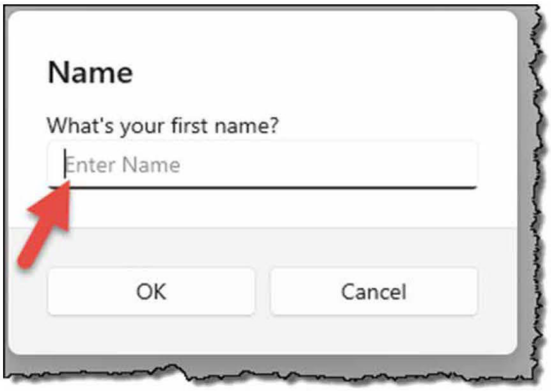
Try It Out

Run the application and click on the **Users** menu. Click on one of the users' delete button and view the alert with the yes/no buttons, as shown in **Figure 5**. You haven't written any delete functionality in your view model class, so this is just an example of how to make the call to the method that would delete data from your data store.

Ask User to Input a Single Value

To ask the user for a single piece of input, call the DisplayPromptAsync() method, as shown in the code snippet below, to display a dialog that looks like **Figure 6**.

```
string response = await Shell.Current
  .CurrentPage.DisplayPromptAsync(
  "Name",
  "What's your first name?");
```



Name

What's your first name?

Enter Name

Figure 7: Add a placeholder to the input area to help the user with what to enter.

```
Console.WriteLine(response);
```

If you click the Cancel button or press the escape key on Windows, a **null** value is returned into the **response** variable. On mobile platforms, you may click on the Cancel button, or just press anywhere outside of the button to return a **null** value into the **response** variable.

Add a Placeholder

The previous example of DisplayPromptAsync() is just one of several overloads and optional parameters you may set on this dialog. For example, if you wish to have placeholder text in the input area (**Figure 7**) pass a string to the **placeholder** argument when calling DisplayPromptAsync(), as shown in the following code:

```
string response = await DisplayPromptAsync(
  "Name",
  "What's your first name?",
  placeholder: "Enter Name");

Console.WriteLine(response);
```

Add an Initial Value

To display an initial value when the pop-up is first displayed, pass a string to the **placeholder** argument when calling DisplayPromptAsync(), as shown in **Figure 8.** The following code shows passing arguments to both the **placeholder** and the **initialValue** parameters. If you delete the initial value, the placeholder is once again displayed within the input area.

```
string response = await DisplayPromptAsync(
  "Name",
  "What's your first name?",
  placeholder: "Enter Name",
  initialValue: "John");

Console.WriteLine(response);
```

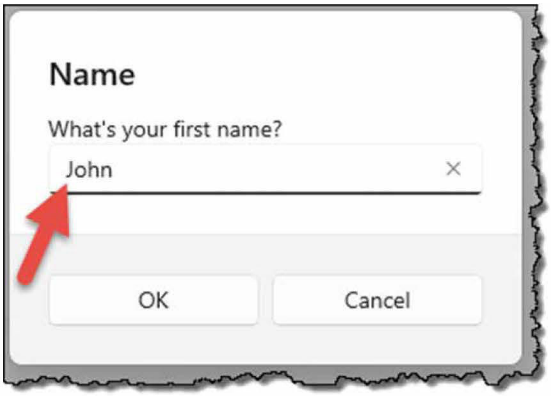
Set MaxLength and Change the Keyboard

You may restrict the number of characters the user is allowed to enter by passing in a numeric value to the **maxLength** parameter. If you are asking a user for numeric input, you may wish to only display the numeric keyboard on mobile devices. The following code passes arguments to both the **maxLength** and the **keyboard** parameters to display the pop-up shown in **Figure 9**.

```
string response = await Shell.Current
   .CurrentPage.DisplayPromptAsync(
    "Age",
    "What's your age?",
    initialValue: "1",
    maxLength:3,
    keyboard: Keyboard.Numeric);

Console.WriteLine(response);
```

Display an Action Sheet

The DisplayActionSheet() method presents one or more options to a user from which they may select a single value. The following code displays a dialog that looks like **Figure 10**.

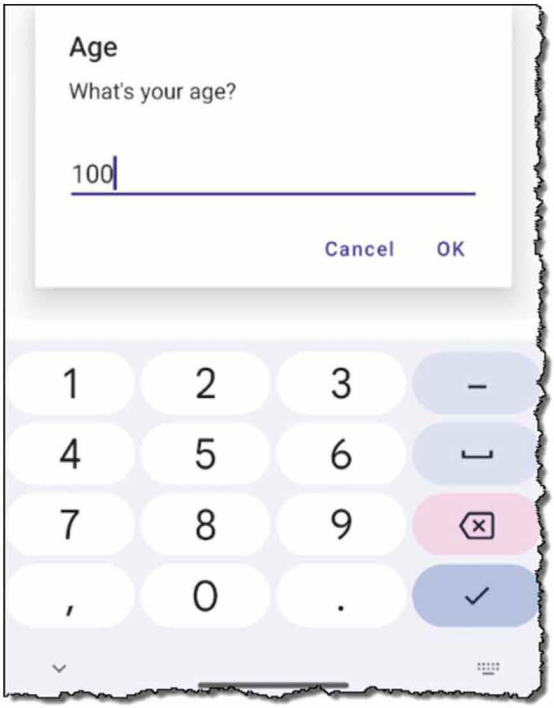


Name

What's your first name?

John

Figure 8: Add an initial value to display when the dialog first appears.

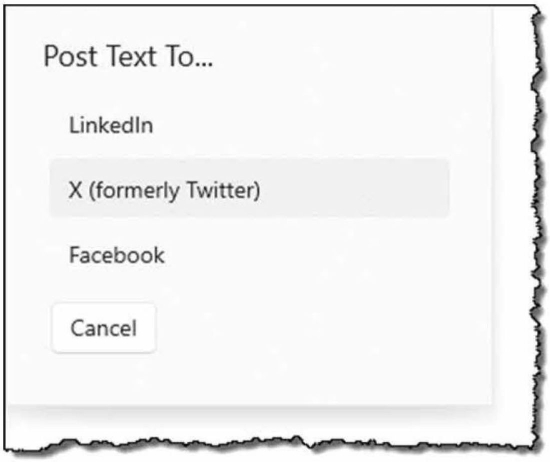


Age

What's your age?

100

Figure 9: The Keyboard property can be set to numeric to display a different keyboard on mobile devices.



Post Text To...

Figure 10: An Action Sheet displays a set of values for the user to select.

```
string action = await Shell.Current
    .CurrentPage.DisplayActionSheet(
    "Post Text To...",
    "Cancel",
    null,
    "LinkedIn",
    "X (formerly Twitter)",
    "Facebook");

Console.WriteLine(action);
```

When the user clicks on one of the values presented, that value is returned and placed into the variable **action**. The **action** variable will be "LinkedIn", "X (formerly Twitter)", or "Facebook". If the Cancel button is pressed, the action variable is set to "Cancel". On Windows, you may press the escape key, which is the same as pressing the Cancel button. On mobile devices, you may press anywhere outside the dialog and that's also the same as pressing the Cancel button.

Display a Second Button on Action Sheet

If you pass a string to the third parameter instead of a null value, that string is presented as a second button (see **Figure 11**) as shown in the following code snippet. This third parameter is known as a destruction button as it's generally used to let the user dispose of something instead of saving something. If this button is pressed, the text of the button is returned in the **action** variable.

```
string action = await Shell.Current
   .CurrentPage.DisplayActionSheet(
   "Save Product Photo?",
   "Cancel",
   "Delete Photo",
   "To Photo Roll",
   "To File",
   "As Email Attachment");

Console.WriteLine(action);
```

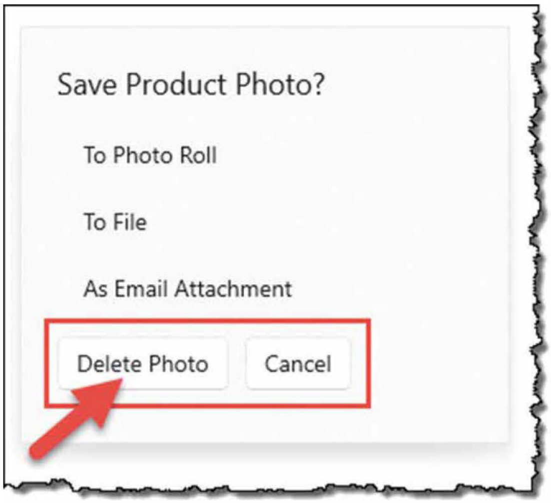


Figure 11: An action sheet may have two buttons if needed.

Action sheets can be dismissed on any touch platform and MacCatalyst by tapping on the page outside of the action sheet. On a Windows desktop, action sheets can be cleared by pressing the escape key, clicking the Cancel button, or by clicking on the page outside the action sheet.

Listing 9: Create two properties to display messages on your pages.

```
public bool IsInfoAreaVisible {
  get { return _IsInfoAreaVisible; }
  set {
    _IsInfoAreaVisible = value;
    RaisePropertyChanged(nameof(IsInfoAreaVisible));
  }
}

public bool IsExceptionAreaVisible {
  get { return _IsExceptionAreaVisible; }
  set {
    _IsExceptionAreaVisible = value;
    RaisePropertyChanged(nameof(IsExceptionAreaVisible));
  }
}
```

Displaying Informational Messages

Throughout the last couple of articles, you've set the **InfoMessage** and the **LastErrorMessage** properties to various strings to display. However, these properties aren't bound anywhere on your pages, so let's add some reusable components to display these different messages. On most pages, you're going to want to have areas where you can display informational, exception, and validation messages. Informational messages are used to inform users about something that's currently happening, or the result of some operation. Exception messages inform the user that some error occurred while attempting to process data. Validation messages inform the user of incorrect data entry on their part.

Add Properties to the View Model Base Class

Before you learn about validation messages, let's first tackle the information and exception messages. Create a couple of Boolean properties you can bind to make the information and exception message areas visible or not. Open the **BaseClasses\ViewModelBase.cs** file in the Common.Library project and add two new private variables, as shown in the following code snippet:

```
private bool _IsInfoAreaVisible;
private bool _IsExceptionAreaVisible;
```

Create two public properties to expose these two private variables, as shown in **Listing 9**. Each of the setters raises the **PropertyChanged** event so when they are set, the different message areas become visible.

In the GetAsync() and GetAsync(id) methods in your view models call the repository class to attempt to get data. Just before this call, invoke a method in the **ViewModelBase** class called BeginProcessing(). Just before the end of each method, call another method named EndProcessing(). Within each of these methods, set the view model properties to reflect the appropriate state at that moment in time. In the BeginProcessing() method, shown below, set the properties to a valid start state just before you call the repository method to load or save data. Add the following method to the **ViewModelBase** class:

```
#region BeginProcessing Method
protected virtual void BeginProcessing() {
  InfoMessage = string.Empty;
  LastErrorMessage = string.Empty;
  LastException = null;
  RowsAffected = 0;
  IsInfoAreaVisible = true;
  IsExceptionAreaVisible = false;
}
#endregion
```

The EndProcessing() method is called after the processing has succeeded or has failed with an exception. This method checks to see if the **LastErrorMessage** is not null or empty, or that the **LastException** property is not null. If either of these conditions are true, then the **IsExceptionAreaVisible** property is set to true. Add the following method to the **ViewModelBase** class:

```
#region EndProcessing Methods
protected virtual void EndProcessing() {
  if (!string.IsNullOrEmpty(LastErrorMessage) ||
      LastException != null) {
    IsExceptionAreaVisible = true;
  }
}
#endregion
```

Modify the User View Model Class

Open the **ViewModelClasses\UserViewModel.cs** file. In the **GetAsync()** method, remove the line of code **RowsAffected = 0;** at the beginning of the method. Call the **BeginProcessing()** method just before the **try** block. Call the **EndProcessing()** method just after the closing brace for the **catch** block, as shown in the following code snippet:

```
public async
  Task<ObservableCollection<User>> GetAsync() {

  BeginProcessing();
  try {
    // REST OF THE CODE HERE
  }
  catch (Exception ex) {
    PublishException(ex);
  }
  EndProcessing();

  return Users;
}
```

In the **GetAsync(id)** method, call the **BeginProcessing()** method just before the **try** block. Call the **EndProcessing()** method just after the closing brace for the **catch** block. Within the **catch** block, remove the **RowsAffected = 0;** line of code, as shown in the following code snippet:

```
public async Task<User?> GetAsync(int id) {
  BeginProcessing();
  try {
    // REST OF THE CODE HERE
  }
  catch (Exception ex) {
    PublishException(ex);
  }
  EndProcessing();

  return CurrentEntity;
}
```

Modify the Product View Model

Modify the product view model class the same as you did in the user view model class. Open the **ViewModelClasses\ProductViewModel.cs** file. In the **GetAsync()** method, remove the line of code **RowsAffected = 0;** at the beginning of the method. Call the **BeginProcessing()** method just before the **try** block. Call the **EndProcessing()** method just after the closing brace for the **catch** block.

In the **GetAsync(id)** method, call the **BeginProcessing()** method just before the **try** block. Call the **EndProcessing()** method just after the closing brace for the **catch** block. Within the **catch** block, remove the **RowsAffected = 0;** line of code. Feel free to update the ColorViewModel class with these changes as well.

Partial Message Pages

To provide the most reusable code, create a set of partial pages to display information, exception, and validation messages. Even though the information and exception partial pages are just a simple Label control, this gives you the flexibility to change the look of the message display on all pages by just changing these in a single location.

Create the Information Message Page

Right mouse-click on the **ViewsPartial** folder and select **Add > New Item… > .NET MAUI > Content View (XAML)**, set the name to **InfoMessageArea**, and click on the Add button. In the <ContentView> starting tag, add a new XML namespace to reference the Common.Library.

```
xmlns:common="clr-namespace:
  Common.Library;assembly=Common.Library"
```

Add an **x:DataType** attribute to the <ContentView> starting tag to bind properties from the **ViewModelBase** class to this content view control.

```
x:DataType="common:ViewModelBase"
```

Replace the <VerticalStackLayout> element with the following XAML. This label control is bound to the **InfoMessage** property. All information messages are displayed by this label.

```
<Label
  IsVisible="{Binding IsInfoAreaVisible}"
  Text="{Binding Path=InfoMessage}"
  Style="{StaticResource InfoMessageArea}" />
```

Open the **Resources\Styles\CommonStyles.xaml** file in the **Common.Library.MAUI** project and add a new **keyed** style, as shown in the following XAML:

```
<Style TargetType="Label"
       x:Key="InfoMessageArea">
  <Setter Property="FontSize"
          Value="18" />
  <Setter Property="Margin"
          Value="4,4" />
</Style>
```

Create the Exception Message Page

Right mouse-click on the **ViewsPartial** folder and select **Add > New Item... > .NET MAUI > Content View (XAML)**, set the name to **ExceptionMessageArea**, and click on the Add button. In the <ContentView>starting tag, add a new XML namespace to reference the Common.Library.

```
xmlns:common="clr-namespace:
  Common.Library;assembly=Common.Library"
```

Add an **x:DataType** attribute to the <ContentView> starting tag to bind properties from the **ViewModelBase** class to this content view control.

```
x:DataType="common:ViewModelBase"
```

Replace the <VerticalStackLayout> element with the following XAML. This label control is bound to the **LastErrorMessage** property. All exception messages are displayed by this label.

```
<Label
  IsVisible="{Binding IsExceptionAreaVisible}"
  Text="{Binding LastErrorMessage}"
  Style="{StaticResource ErrorMessage}" />
```

Open the **Resources\Styles\CommonStyles.xaml** file in the **Common.Library.MAUI** project and add a new **keyed** style, as shown in the following XAML:

```
<Style TargetType="Label"
       x:Key="ErrorMessage">
  <Setter Property="FontAttributes"
          Value="Bold" />
  <Setter Property="TextColor"
          Value="Red" />
</Style>
```

Modify the List Views to Display Informational Messages

It's now time to add these two new partial views to the pages you've created so far. The information message area is going to be displayed just below the reusable header view. Open the **Views\UserListView.xaml** file and add a new row definition to the **RowDefinitions** attribute so it looks like the XAML shown below:

```
<Grid.RowDefinitions>
  <RowDefinition Height="Auto" />
  <RowDefinition Height="Auto" />
  <RowDefinition Height="*" />
</Grid.RowDefinitions>
```

In between the <partial:HeaderView ...> and the <ListView> ,add the Information Message Partial Page.

```
<!-- *********************** -->
<!-- Information Message Area -->
<!-- *********************** -->
<partial:InfoMessageArea Grid.Row="1"
  Grid.ColumnSpan="2" />
```

Because you inserted the information message control before the ListView control, modify the **Grid.Row** property on the ListView control to "2", as shown in the following code snippet:

```
<ListView Grid.Row="2" ...>
```

Modify the Product List View

Open the **Views\ProductListView.xaml** file and add a new row definition so the **RowDefinitions** looks like the XAML shown below.

```
<Grid.RowDefinitions>
  <RowDefinition Height="Auto" />
  <RowDefinition Height="Auto" />
  <RowDefinition Height="*" />
</Grid.RowDefinitions>
```

In between the <partial:HeaderView ...> and the <CollectionView>, add the Information Message Partial Page.

```
<!-- *********************** -->
<!-- Information Message Area -->
<!-- *********************** -->
<partial:InfoMessageArea Grid.Row="1"
  Grid.ColumnSpan="2" />
```

Because you inserted the information message control before the CollectionView control, modify the **Grid.Row** property on the CollectionView control to "2", as shown in the following code snippet:

```
<CollectionView Grid.Row="2" ...>
```

Try It Out

Run the application, click on the **Users** menu and you should see the informational message "Found 3 Users" displayed. Click on the **Products** menu and see the informational message "Found 4 Products" displayed.

Display Processing Messages

Open the **ViewModelClasses\UserViewModel.cs** file and add in the GetAsync() method. Just before the call to **await Repository.GetAsync()**, add the following informational message:

```
InfoMessage = "Please wait while loading users.";
```

To simulate this process taking a long time, temporarily add an additional line just **below** this message.

```
await Task.Run(() => Thread.Sleep(2000));
```

Open the **ViewModelClasses\ProductViewModel.cs** file and add in the GetAsync() method, just before the await Repository.GetAsync() call, add the following informational message.

```
InfoMessage = "Please wait while loading products.";
```

Try It Out

Run the application, click on the **Users** menu, and you should see informational message **Please wait while loading users.** displayed. After about two seconds, this message is removed, and the **Found 3 Users** message should appear. Remove the **Thread.Sleep()** call in the UsersViewModel class.

Display Error Messages

Open the **Views\UserListView.xaml** file and add a new row definition so the RowDefinitions looks like the XAML shown below:

```
<Grid.RowDefinitions>
  <RowDefinition Height="Auto" />
  <RowDefinition Height="Auto" />
  <RowDefinition Height="Auto" />
  <RowDefinition Height="*" />
</Grid.RowDefinitions>
```

Insert the exception message partial page just after the information message area.

```
<!-- ***************** -->
<!-- Error Message Area -->
<!-- ***************** -->
<partial:ExceptionMessageArea
  Grid.Row="2"
  Grid.ColumnSpan="2" />
```

Change the **Grid.Row** property on the <ListView> to "3".

```
<ListView Grid.Row="3" ...>
```

Modify the Product List View

Open the **Views\ProductListView.xaml** file and add a new row definition so the RowDefinitions looks like the XAML shown below:

```
<Grid.RowDefinitions>
  <RowDefinition Height="Auto" />
  <RowDefinition Height="Auto" />
  <RowDefinition Height="Auto" />
  <RowDefinition Height="*" />
</Grid.RowDefinitions>
```

Insert the exception message partial page just after the information message area.

```
<!-- ***************** -->
<!-- Error Message Area -->
<!-- ***************** -->
<partial:ExceptionMessageArea
  Grid.Row="2"
  Grid.ColumnSpan="2" />
```

Change the **Grid.Row** property on the <CollectionView> to "3".

```
<CollectionView Grid.Row="3" ...>
```

Open the **ExtensionClasses\ServiceExtensions.cs** file and, in the AddRepositoryClasses() method, comment out the line that adds the **UserRepository** to the DI container.

```
// services.AddScoped<IRepository<User>,
   UserRepository>();
```

Try It Out

Run the application and click on the **Users** menu and, toward the bottom of the page, you should see the error message **The Repository Object is not Set** displayed. Open the **ExtensionClasses\ServiceExtensions.cs** file and restore the line you previously commented so you won't get that error again.

Display Error Messages on Detail Views

Open the **Views\UserDetailView.xaml** file and add a new **Auto** to the RowDefinition attribute. Scroll down to the bottom of the file, locate the </HorizontalStackLayout> closing tag, and insert the following XAML in between this element and the </Grid> closing tag.

```
<!-- ***************** -->
<!-- Error Message Area -->
<!-- ***************** -->
<partial:ExceptionMessageArea
  Grid.Row="12"
  Grid.ColumnSpan="2" />
```

Modify Product Detail View

Open the **Views\ProductDetailView.xaml** file and add a new **Auto** to the RowDefinition attribute. Scroll down to the bottom of the file, locate the </HorizontalStackLayout> closing tag, and insert the following XAML in between this element and the </Grid> closing tag:

```
<!-- ***************** -->
<!-- Error Message Area -->
<!-- ***************** -->
<partial:ExceptionMessageArea
  Grid.Row="16"
  Grid.ColumnSpan="2" />
```

Validate Data Using Data Annotations

Data annotations are not just for use in ASP.NET web applications. Any type of .NET application can use data annotations for validating data. It only takes about 10 lines of code to programmatically validate data annotations attached to entity classes. There are many built-in data annotations supplied by Microsoft to validate your data quickly. For more information on using data annotations, see my article entitled **The Rich Set of Data Annotation and Validation Attributes in .NET** in the January/February 2023 issue of CODE Magazine at https://tinyurl.com/tyee979v.

Add Data Annotations to Entity Class

Let's add some data annotations to the User class to enforce business rules, such as which fields are required, have a maximum and/or minimum string length, or have a range of values it must fall within. Open the **EntityClasses\User.cs** file in the AdventureWorks.EntityLayer project and replace the entire file with the code shown in **Listing 10**.

Microsoft Data Annotations

Instead of writing validation code in your view model class as you did in **Listing 1**, add attributes above those properties in your **User** class that you wish to validate. There are many standard data annotation attributes supplied by Microsoft such as [Required], [MinLength], [MaxLength], [StringLength], and [Range]. From the name of the attribute, you can infer what each of these attributes validates on a property. For a complete list of data annotation attributes, visit Microsoft's website at https://bit.ly/3TJICid.

Getting the Sample Code

You can download the sample code for this article by visiting www.CODEMag.com under the issue and article, or by visiting www.pdsa.com/downloads. Select "Articles" from the Category drop-down. Then select "Exploring .NET MAUI: Popups, Messages, and Data Validation" from the Item drop-down.

Listing 10: Add data annotations to your class to enforce business rules on the data entered by the user.

```
using System.ComponentModel.DataAnnotations.Schema;
using System.ComponentModel.DataAnnotations;
using Common.Library;

namespace AdventureWorks.EntityLayer;

/// <summary>
/// This class contains properties that
/// map to each field in the dbo.User table.
/// </summary>
[Table("User", Schema = "dbo")]
public partial class User : EntityBase {
  #region Private Variables
  private int _UserId;
  private string _LoginId = string.Empty;
  private string _FirstName = string.Empty;
  private string _LastName = string.Empty;
  private string _Email = string.Empty;
  private string _Password = "P@ssW0Rd";
  private string _Phone = string.Empty;
  private string? _PhoneType;
  private bool? _IsFullTime;
  private bool? _IsEnrolledIn401k;
  private bool? _IsEnrolledInHealthCare;
  private bool? _IsEnrolledInHSA;
  private bool? _IsEnrolledInFlexTime;
  private bool? _IsEmployed;
  private DateTime? _BirthDate;
  private TimeSpan? _StartTime;
  #endregion

  #region Public Properties
  [Display(Name = "User Id")]
  [Required(ErrorMessage = "{0} must be filled in.")]
  public int UserId {
    get { return _UserId; }
    set {
      _UserId = value;
      RaisePropertyChanged(nameof(UserId));
    }
  }

  [Display(Name = "Login Id")]
  [Required(ErrorMessage = "{0} must be filled in.")]
  [StringLength(20, MinimumLength = 0,
    ErrorMessage = "{0} must be between {2}
                     and {1} characters long.")]
  public string LoginId {
    get { return _LoginId; }
    set {
      _LoginId = value;
      RaisePropertyChanged(nameof(LoginId));
    }
  }

  [Display(Name = "First Name")]
  [Required(ErrorMessage = "{0} must be filled in.")]
  [StringLength(50, MinimumLength = 0,
    ErrorMessage = "{0} must be between {2}
                    and {1} characters long.")]
  public string FirstName {
    get { return _FirstName; }
    set {
      _FirstName = value;
      RaisePropertyChanged(nameof(FirstName));
    }
  }

  [Display(Name = "Last Name")]
  [Required(ErrorMessage = "{0} must be filled in.")]
  [StringLength(50, MinimumLength = 0,
    ErrorMessage = "{0} must be between {2}
                    and {1} characters long.")]
  public string LastName {
    get { return _LastName; }
    set {
      _LastName = value;
      RaisePropertyChanged(nameof(LastName));
    }
  }

  [Display(Name = "Email")]
  [Required(ErrorMessage = "{0} must be filled in.")]
  [StringLength(256, MinimumLength = 0,
    ErrorMessage = "{0} must be between {2}
                    and {1} characters long.")]
  [EmailAddress()]
  public string Email {
    get { return _Email; }
    set {
      _Email = value;
      RaisePropertyChanged(nameof(Email));
    }
  }

  [Display(Name = "Password")]
  [Required(ErrorMessage = "{0} must be filled in.")]
 [StringLength(50, MinimumLength = 0,
   ErrorMessage = "{0} must be between {2}
                   and {1} characters long.")]
  public string Password {
    get { return _Password; }
    set {
      _Password = value;
      RaisePropertyChanged(nameof(Password));
    }
  }

  [Display(Name = "Phone")]
  [Required(ErrorMessage = "{0} must be filled in.")]
  [StringLength(25, MinimumLength = 0,
    ErrorMessage = "{0} must be between {2}
                    and {1} characters long.")]
  [Phone()]
  public string Phone {
    get { return _Phone; }
    set {
      _Phone = value;
      RaisePropertyChanged(nameof(Phone));
    }
  }

  [Display(Name = "Phone Type")]
  [StringLength(10, MinimumLength = 0,
    ErrorMessage = "{0} must be between {2}
                    and {1} characters long.")]
  public string? PhoneType {
    get { return _PhoneType; }
    set {
      _PhoneType = value;
      RaisePropertyChanged(nameof(PhoneType));
    }
  }

  [Display(Name = "Is Full Time")]
  public bool? IsFullTime {
    get { return _IsFullTime; }
    set {
      _IsFullTime = value;
      RaisePropertyChanged(nameof(IsFullTime));
```

Listing 10: continued...

```
    }
  }

  [Display(Name = "Is Enrolled In 40 1K")]
  public bool? IsEnrolledIn401k {
    get { return _IsEnrolledIn401k; }
    set {
      _IsEnrolledIn401k = value;
      RaisePropertyChanged(nameof(IsEnrolledIn401k));
    }
  }

  [Display(Name = "Is Enrolled In Health Care")]
  public bool? IsEnrolledInHealthCare {
    get { return _IsEnrolledInHealthCare; }
    set {
      _IsEnrolledInHealthCare = value;
      RaisePropertyChanged(nameof(IsEnrolledInHealthCare));
    }
  }

  [Display(Name = "Is Enrolled In Hsa")]
  public bool? IsEnrolledInHSA {
    get { return _IsEnrolledInHSA; }
    set {
      _IsEnrolledInHSA = value;
      RaisePropertyChanged(nameof(IsEnrolledInHSA));
    }
  }

  [Display(Name = "Is Enrolled In Flex Time")]
  public bool? IsEnrolledInFlexTime {
    get { return _IsEnrolledInFlexTime; }
    set {
      _IsEnrolledInFlexTime = value;
      RaisePropertyChanged(nameof(IsEnrolledInFlexTime));
    }
  }

  [Display(Name = "Is Employed")]
  public bool? IsEmployed {
    get { return _IsEmployed; }
    set {
      _IsEmployed = value;
      RaisePropertyChanged(nameof(IsEmployed));
    }
  }

  [Display(Name = "Birth Date")]
  public DateTime? BirthDate {
    get { return _BirthDate; }
    set {
      _BirthDate = value;
      RaisePropertyChanged(nameof(BirthDate));
    }
  }

  [Display(Name = "Start Time")]
  public TimeSpan? StartTime {
    get { return _StartTime; }
    set {
      _StartTime = value;
      RaisePropertyChanged(nameof(StartTime));
    }
  }

  public string FullName {
    get { return FirstName + " " + LastName; }
  }

  public string LastNameFirstName {
    get { return LastName + ", " + FirstName; }
  }
  #endregion

  #region ToString Override
  public override string ToString() {
    return $"{LastName}";
  }
  #endregion
}
```

The [Required] Attribute

The various [Required] attributes shown in **Listing 10** do exactly what you think: It ensures that the user has filled in some data in the input field bound to the property that this attribute decorates. Each of the data annotation attributes is inherited from an abstract class named **ValidationAttribute**. This validation attribute class has properties that each of the inherited attribute classes can use. These properties are shown in **Table 1** and that you see in several in the attributes in **Listing 10**.

Property Name	Description
ErrorMessage	Get/Set the error message format string
ErrorMessageString	Gets the localized error message
ErrorMessageResourceName	Get/Set the error message resource name
ErrorMessageResourceType	Get/Set the error resource class type

Table 1: The common properties available to all data annotation attribute classes

The ErrorMessage Property

The **ErrorMessage** property is what you use to report back the error message to the user. Use a hard-coded string or use placeholders in the string to automatically retrieve data from the property the attribute is decorating. The placeholders are what the String.FormatString() method uses where you add numbers enclosed within curly braces, as shown in the following [Required] attribute:

```
[Required(ErrorMessage = "{0} Must Be Filled In")]
public string FirstName { get; set; }
```

The {0} placeholder is replaced with the name of the property the attribute is decorating. In the above example, the resulting string is "FirstName Must Be Filled In". Next, look at the following code snippet that uses the [Range] attribute.

```
[Range(0.01, 9999,
  ErrorMessage = "{0} must be between
                  {1} and {2}")]
public decimal? Price { get; set; }
```

The {1} placeholder is replaced with the value in the first parameter to the **Range** attribute and the {2} placeholder is replaced with the value in the second parameter. If you have more parameters, you keep on incrementing the placeholder accordingly.

The [DisplayName] Attribute

Displaying the property name to the user is generally not a good idea. Sometimes the property name won't mean much to the user. It's better to use a more readable string, such as the same label displayed on an input form. You can accomplish this by adding the [DisplayName] attribute to any property in your class. In **Listing 10**, every property has a [DisplayName] attribute so it can have a friendly label to display to the user if the property fails validation. If the [DisplayName] attribute is attached to a property, the {0} placeholder in the **ErrorMessage** property uses the **Name** property from the [DisplayName] attribute instead of the actual property name.

Listing 11: Create a class to hold the validation message to display to the user.

```
namespace Common.Library;

public class ValidationMessage : CommonBase {
  #region Private Variables
  private string _PropertyName = string.Empty;
  private string _Message = string.Empty;
  #endregion

  #region Public Properties
  public string PropertyName {
    get { return _PropertyName; }
    set {
      _PropertyName = value;
      RaisePropertyChanged(nameof(PropertyName));
    }
  }

  public string Message {
    get { return _Message; }
    set {
      _Message = value;
      RaisePropertyChanged(nameof(Message));
    }
  }
  #endregion

  #region ToString() Override
  public override string ToString() {
    if (string.IsNullOrEmpty(PropertyName)) {
      return $"{Message}";
    }
    else {
      return $"{Message} ({PropertyName})";
    }
  }
  #endregion
}
```

Listing 12: Add two properties to display validation messages to the user.

```
public bool IsValidationAreaVisible {
  get { return _IsValidationAreaVisible; }
  set {
    _IsValidationAreaVisible = value;
    RaisePropertyChanged(
      nameof(IsValidationAreaVisible));
  }
}

public ObservableCollection<ValidationMessage>
  ValidationMessages {
  get { return _ValidationMessages; }
  set {
    _ValidationMessages = value;
    RaisePropertyChanged(
      nameof(ValidationMessages));
  }
}
```

Other Attributes

Other attributes you see in **Listing 10** are **StringLength**, **EmailAddress**, and **Phone**. **StringLength**. These attributes ensure that a user enters a string value between the minimum and maximum lengths specified. The **EmailAddress** and **Phone** attributes validate that the user entered a valid email address and valid phone number, respectively.

Add Validation Classes to the Common Library

To display a list of which fields don't pass the validation tests specified by the various attributes, create a class to hold the property name that failed and the validation message to display to the user. Right mouse-click on the **Common.Library** project and add a new folder named **ValidationClasses**. Right mouse-click on the **ValidationClasses** folder and add a new class named **ValidationMessage**. Replace the entire contents of this new file with the code shown in **Listing 11**.

Add to the view model base class a Boolean property of whether to display the validation message area, and a collection of **ValidationMessage** objects. Open the **BaseClasses\ViewModelBase.cs** file and add two private variables.

```
private bool _IsValidationAreaVisible;
private ObservableCollection<ValidationMessage>
   _ValidationMessages = new();
```

Next, add the two public properties (**Listing 12**) that you can bind to controls on a page. Locate the BeginProcessing() method and initialize both of these new properties.

```
IsValidationAreaVisible = false;
ValidationMessages.Clear();
```

Locate the EndProcessing() method and check to see if any validation errors have occurred and set the **IsValidationAreaVisible** property appropriately.

```
IsValidationAreaVisible =
   ValidationMessages.Count > 0;
```

Also in this class, add a generic Validate() method (**Listing 13**) to which you pass any entity class. This method checks the values in each property against any data annotations on that property. The code in **Listing 13** creates an instance of a **ValidationContext** object passing in the **Entity** property. Create an instance of list of **ValidationResult** objects so the TryValidateObject() method can fill in this list with all the **ValidationResult** objects. The TryValidateObject() method is responsible for checking all data annotations attached to each property in the entity object. If any validations fail, the appropriate error message, along with the property name, is returned in the **results** variable.

Loop through the **results** collection and add a new **ValidationMessage** object to the **ValidationMessages** property. The **ErrorMessage** property is filled in with the **ErrorMessage** property from the current ValidationResult item. The property name is retrieved from the first element of the

Listing 13: Use the ValidationContext and Validator objects to validate properties decorated with data annotations.

```
#region Validate Method
public bool Validate<T>(T entity) {
  ValidationMessages.Clear();

  if (entity != null) {
    // Create ValidationContext object
    ValidationContext context = new(entity,
      serviceProvider: null, items: null);
    List<ValidationResult> results = new();

    // Call TryValidateObject() method
    if (!Validator.TryValidateObject(
        entity, context, results, true)) {
      // Get validation results
      foreach (ValidationResult item in results) {
        string propName = string.Empty;
        if (item.MemberNames.Any()) {
          propName =
            ((string[])item.MemberNames)[0];
        }
        // Build new ValidationMessage object
        ValidationMessage msg = new() {
          Message = item.ErrorMessage
            ?? string.Empty,
          PropertyName = propName
        };

        // Add validation object to list
        ValidationMessages.Add(msg);
      }
    }
  }

  IsValidationAreaVisible
    = ValidationMessages.Count > 0;

  return !IsValidationAreaVisible;
}
#endregion
```

MemberNames property on the **ValidationResult** item. It's possible for a data annotation to have two properties to which it applies, but for most simple properties, you only need to grab the first property name.

Add Validate() Method to User View Model Class

Before you save the **User** object with the data entered by the user, you need to validate that data. Open the **ViewModelClasses\UserViewModel.cs** file and modify the SaveAsync() method to call the Validate() method, as shown in the following code:

```
public async virtual Task<User?> SaveAsync() {
  User? ret;

  if (Validate(CurrentEntity)) {
    // TODO: Write code to save data

    ret = new User();
  }
  else {
    ret = null;
  }

  return await Task.FromResult(ret);
}
```

Display Validation Errors

Just like you did with the information and exception messages, create a validation error partial page. Right mouse-click on the **ViewsPartial** folder and select **Add > New Item… > .NET MAUI > Content View (XAML)**, set the name to **ValidationMessageArea**, and click on the Add button. In the <ContentView> starting tag, add a new XML namespace to reference the Common.Library.

```
xmlns:common="clr-namespace:
  Common.Library;assembly=Common.Library"
```

Add an **x:DataType** attribute to this starting tag so you can bind to the two properties you created in the **ValidationMessage** class, as shown in the following code:

```
x:DataType="common:ViewModelBase"
```

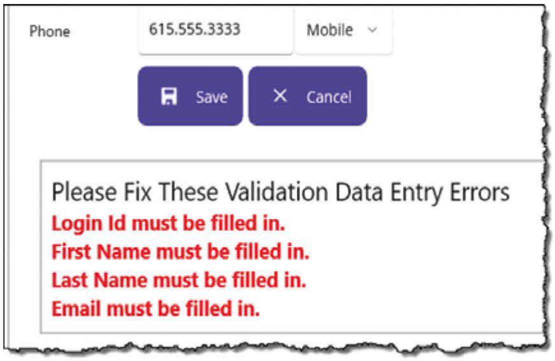
Replace the <VerticalStackLayout> element with the XAML shown in **Listing 14**. It's in this **CollectionView** list where you display the validation messages from any

Listing 14: Use a CollectionView to display validation error messages.

```
<Border
   IsVisible="{Binding IsValidationAreaVisible}"
   Style="{StaticResource ValidationErrorArea}">
  <VerticalStackLayout>
    <Label FontSize="Medium"
      Text="Please Fix These Validation
            Data Entry Errors" />
    <CollectionView
      ItemsSource="{Binding ValidationMessages}">
      <CollectionView.ItemTemplate>
        <DataTemplate
          x:DataType="common:ValidationMessage">
          <HorizontalStackLayout>
            <Label Text="{Binding Message}"
               Style="{StaticResource
                       ValidationMessage}" />
          </HorizontalStackLayout>
        </DataTemplate>
      </CollectionView.ItemTemplate>
    </CollectionView>
  </VerticalStackLayout>
</Border>
```

Listing 15: Add two keyed styles for displaying the validation errors.

```
<Style TargetType="Label"
       x:Key="ValidationMessage">
  <Setter Property="FontAttributes"
          Value="Bold" />
  <Setter Property="FontSize"
          Value="Small" />
  <Setter Property="TextColor"
          Value="Red" />
</Style>
<Style TargetType="Border"
       x:Key="ValidationErrorArea">
  <Setter Property="Stroke"
          Value="Gray" />
  <Setter Property="Margin"
          Value="10" />
  <Setter Property="Padding"
          Value="10" />
</Style>
```



Please Fix These Validation Data Entry Errors
Login Id must be filled in.
First Name must be filled in.
Last Name must be filled in.
Email must be filled in.

Figure 12: Validation errors are displayed in a colored font to make them stand out.

failed data validation. Open the **Resources\Styles\CommonStyles.xaml** file in the **Common.Library.MAUI** project and add two new **keyed** style, as shown in **Listing 15**.

Modify the Detail View

Open the **Views\UserDetailView.xaml** file and add a new **Auto** to the **RowDefinitions** attribute of the <Grid>. Scroll down to </Grid> closing tag and just before this closing tag, add the validation message partial page, as shown in the following code snippet:

```
<!-- ******************************** -->
<!-- Validation Failure Messages Area -->
<!-- ******************************** -->
<partial:ValidationMessageArea
  Grid.Row="13"
  Grid.ColumnSpan="2" />
```

Try It Out

Run the application and click on the **Users** menu. Click the **Edit** button for one of the users. Delete the values in the Login ID, First Name, Last Name, and Email Address entry controls. Click the **Save** button and you should see a list of validation errors appear just below the Save button, as shown in **Figure 12**.

Move Message Area Views to Common MAUI Library

Thinking ahead, you may want the information, error, and validation message components to be used across all .NET MAUI applications you develop. If that's the case, move these partial views to the **Common.Library.MAUI** class library. When moving XAML components from one assembly to another, you have a couple of items to change. Let's look at how to move these files.

The partial views use the **ViewModelBase** class located in the Common.Library project, so set a reference to this class library from the Common.Library.MAUI project. Right mouse-click on the Dependencies folder in the Common.Library.MAUI project and add a project reference to the Common.Library project. Right mouse-click on the Common.Library.MAUI project and add a new folder named **ViewsPartial**.

Move the ExceptionMessageArea.xaml, the InfoMessageArea.xaml, and the ValidationMessageArea.xaml files to this new folder. Open the **ExceptionMessageArea.xaml.cs** file and change the beginning part of the namespace from **AdventureWorks** to **Common.Library** as shown in the following line:

```
namespace Common.Library.MAUI.ViewsPartial;
```

Open the **ExceptionMessageArea.xaml** file and locate the **x:Class** attribute on the ContentView starting tag and change the beginning part of the namespace to **Common.Library** as well.

```
x:Class="Common.Library.MAUI
  .ViewsPartial.ExceptionMessageArea"
```

Open the **InfoMessageArea.xaml.cs** file and change the beginning part of the namespace from **AdventureWorks** to **Common.Library**, as shown in the following line:

```
namespace Common.Library.MAUI.ViewsPartial;
```

Open the **InfoMessageArea.xaml** file and locate the **x:Class** attribute on the ContentView starting tag and change the beginning part of the namespace to **Common.Library** as well.

```
x:Class="Common.Library.MAUI
  .ViewsPartial.InfoMessageArea"
```

Open the **ValidationMessageArea.xaml.cs** file and change the beginning part of the namespace from

ADVERTISERS INDEX

Advertisers Index

CODE Consulting--AI Services
www.codemag.com/ai-services 2

CODE Consulting--Executive Briefing
www.codemag.com/executivebriefing 71

CODE Consulting
www.codemag.com/Code 75

CODE Staffing
www.codemag.com/staffing 76

DevIntersection
www.devintersection.com 59

dtSearch
www.dtSearch.com 15

Ironpdf
www.ironpdf.com 5

Advertising Sales:
Tammy Ferguson
832-717-4445 ext 26
tammy@codemag.com

This listing is provided as a courtesy to our readers and advertisers.
The publisher assumes no responsibility for errors or omissions.

dtSearch IRONSOFTWARE CODE STAFFING next GenAI CONFERENCE

AdventureWorks to **Common.Library**, as shown in the following line:

```
namespace Common.Library.MAUI.ViewsPartial;
```

Open the **ValidationMessageArea.xaml** file and locate the **x:Class** attribute on the ContentView starting tag and change the beginning part of the namespace to **Common.Library** as well.

```
x:Class="Common.Library.MAUI
  .ViewsPartial.ValidationMessageArea"
```

Update References on Application Pages

Now that you've moved these partial pages to a different project, you need to modify the references on the four pages you've used these on. Open the **ProductDetailView.xaml** file and add a new XML namespace to reference the Common.Library.MAUI project.

```
xmlns:commonPartial="clr-namespace:
  Common.Library.MAUI.ViewsPartial;
  assembly=Common.Library.MAUI"
```

Locate the <partial:ExceptionMessageArea …> element and change **partial** to **commonPartial**, as shown in the following XAML:

```
<commonPartial:ExceptionMessageArea
  Grid.Row="16"
  Grid.ColumnSpan="2" />
```

Open the **ProductListView.xaml** file and add a new XML namespace to reference the Common.Library.MAUI project.

```
xmlns:commonPartial="clr-namespace:
  Common.Library.MAUI.ViewsPartial;
  assembly=Common.Library.MAUI"
```

Locate the <partial:InfoMessageArea …> element and change **partial** to **commonPartial**, as shown in the following XAML:

```
<commonPartial:InfoMessageArea
  Grid.Row="1"
  Grid.ColumnSpan="2" />
```

Locate the <partial:ExceptionMessageArea …> element and change **partial** to **commonPartial**, as shown in the following XAML:

```
<commonPartial:ExceptionMessageArea
  Grid.Row="2"
  Grid.ColumnSpan="2" />
```

Repeat the steps above for the UserDetailView.xaml and UserListView.xaml files. When you're done making all these changes, select **Build > Clean Solution** from the Visual Studio menu. You may also need to close Visual Studio and reopen the project to make sure all the changes are recognized. Sometimes Visual Studio gets confused when you move components around.

Try It Out

Run the application and check you can see the information, error, and validate messages.

Summary

In this article, you created your own modal page and called it from the Login page. You returned the value selected from the modal page back to the calling page using the [QueryProperty] attribute. A class to hold global application settings is something you'll probably create in each application you build. In this article, you built an AppSettings class to hold a Boolean value. There are a few different built-in popup dialogs available in .NET MAUI that you may use. You used the Alert dialog to ask the user if they wish to delete a specific row. Finally, you used Microsoft Data Annotations in your User class to help with validation. You then created a few reusable components to display information, error, and validation messages to the user. I hope you have enjoyed my series on .NET MAUI. There are many more topics to explore. and I hope you continue to use this great product.

Paul D. Sheriff

SPONSORED SIDEBAR:

Stay Connected with CODE Magazine on LinkedIn

Join our professional community on LinkedIn to stay ahead in the tech world.

Why Connect with Us?

Exclusive Content: Get the latest tech insights and articles.

Industry Updates: Stay informed with cutting-edge news.

Networking: Connect with peers and industry leaders.

Help us grow by sharing with colleagues and friends.

Follow us today! www.linkedin.com/company/code-magazine

ONLINE QUICK ID 2505041

Get on the Bus— The Azure Service Bus

Azure Service Bus is Microsoft's cloud offering for developing message-based applications. In this article, I'll explore some fundamental Azure Service Bus techniques like how to configure queues, how to send and receive messages, and how to make sure you're properly handing exceptions and poison messages. But first, let's take a moment to think about why you'd want to build a message-based application in the first place.

AZURE SERVICE BUS

Matthew Hess
csharpartisan@gmail.com
YouTube: @csharpartisan

Matthew Hess is a software engineer from Albuquerque, NM. Matthew got his start programming using Borland's Delphi IDE and has been working with C# and the .NET Framework since version 2. Matthew has developed software for many different industries from legal to financial services, to biotechnology, to non-profit. Matthew currently works for CG Infinity, a Dallas-based software consulting firm. His YouTube Channel, @CSharpAristan, is meant to help programmers of all levels become better at the craft of programming in C#.

What's a Message-Based Architecture For?

The purpose of a message-based architecture is to **decouple the components** of a system. What does that really mean? To see what decoupling is all about, let's first look at the opposite: a synchronous, tightly coupled system.

A Tightly Coupled System

Imagine you're feeling hungry. You walk over to a nearby vending machine, put in some money, and push a button. Voila! The vending machine gives you your snack. This is a traditional, tightly coupled call sequence, as shown in **Figure 1**.

If you think of this in terms of software, perhaps the vending machine is a web service that exposes an "OrderSnack" endpoint. Your customer interaction with the web service is tightly coupled and synchronous. You POST a request message to the service and get back a response, your snack (or maybe a "400: Snack not found" error).

A Loosely Coupled, Message-Based System

Compare this to the following sequence: now you are really hungry, so you walk into your favorite cafe, go to the counter, and order a full breakfast. Back in the kitchen, someone prepares your meal and then someone brings it out to you. In this sequence, the process of placing an order and preparing the order are decoupled and are connected by a message, as shown in **Figure 2**.

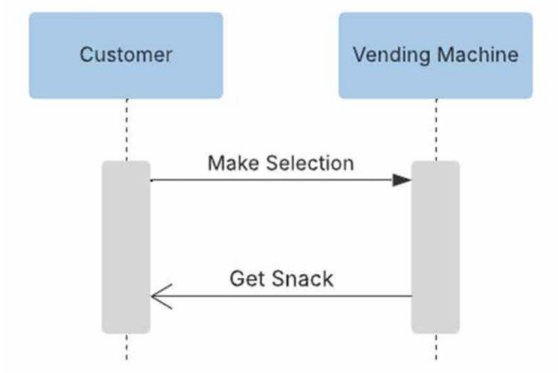


Figure 1: A tightly coupled call sequence

The key feature of this diagram is the presence of the **Order Queue**. This queue decouples the Counter and the Kitchen. The Counter doesn't cook your food or even interact with the Kitchen directly. Instead, the Counter enqueues the order and later a cook in the Kitchen dequeues it and starts preparing it. (In a real kitchen, there's likely a second "delivery queue," i.e., a shelf with heat lamps and a bell, to signal that there is food ready to be brought out).

This decoupling allows the Counter and the Kitchen to be implemented and scaled separately. In the real world, the people you hire for the Counter and the Kitchen do totally different jobs, have different skills and require different equipment. It makes sense to "implement," i.e., staff and equip, these departments separately. Likewise, if the Kitchen is falling behind, you can "scale" it, i.e., hire more cooks, without having to hire more counter clerks. Overall, the system is much more flexible and scalable and can efficiently serve a nice hot meal to many more people than a set of vending machines.

Software systems are filled with these kinds of queue-based systems. Think of an e-commerce platform: the place order, pick-and-pack, ship, and invoice functions are all going to be chained together with messages, not tightly coupled. Azure Service Bus allows you to develop this kind of architecture. It provides the plumbing—the queues and messages—that make a messaging-based architecture possible.

Getting Set Up

Before you can start sending and receiving messages, you need to get a few things ready. Specifically, you need to:

1. Create an Azure Service Bus **Namespace**
2. Create an Azure Service Bus Queue
3. Install the **Azure.Messaging.ServiceBus** NuGet Package in Visual Studio.
4. Connect to the Service Bus Namespace using a **ServiceBusClient** instance.

Create an Azure Service Bus Namespace

An Azure Service Bus **Namespace** is a container that holds one or more **Queues**, **Topics,** and **Subscriptions**. Before you create the first **Queue**, you need to provision a **Namespace** for it to reside in. The Azure CLI code below assumes that you already have an active Azure subscription and have created a resource group.

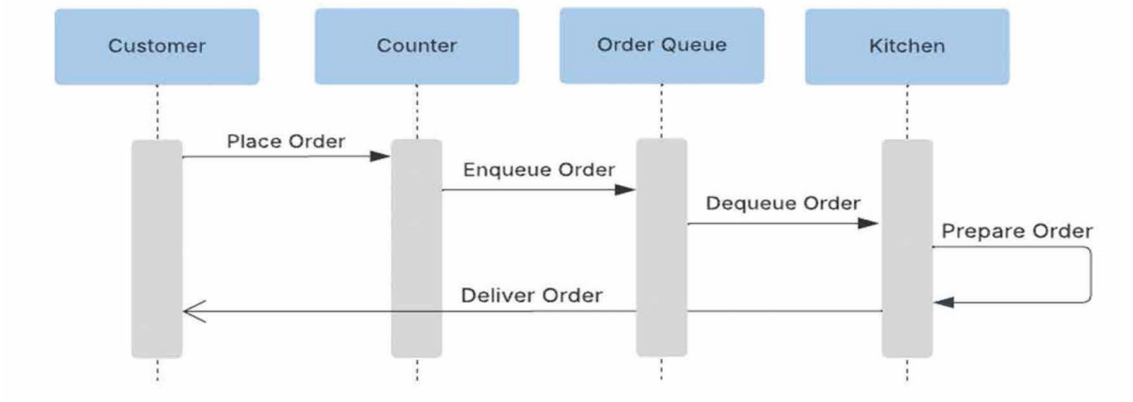
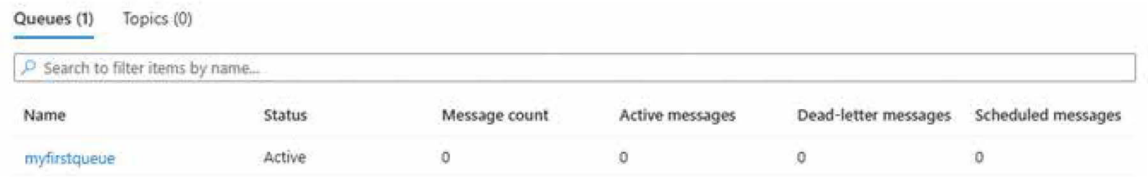


Figure 2: A decoupled, message-based call sequence



Name	Status	Message count	Active messages	Dead-letter messages	Scheduled messages
myfirstqueue	Active	0	0	0	0

Figure 3: The new queue, shown on the Azure Portal

To create an Azure Service Bus namespace, use an Azure CLI command like this

```
az servicebus namespace create `
   --resource-group Resource_Group_1 `
   --name TestBusNamespace `
   --location westus
```

Create an Azure Service Bus Queue

Once you've created the Azure Service Bus Namespace, you can create your first Azure Service Bus Queue using an Azure CLI command like this:

```
az servicebus queue create `
   --resource-group Resource_group_1 `
   --namespace-name TestBusNamespace `
   --name MyFirstQueue
```

Looking in the Azure Portal, you can see how the newly created queue looks (**Figure 3**).

Note that the queue is immediately active and has no messages. You'll soon fix that that by sending your first message!

Install the NuGet Package

All of the Azure Service Bus code will reply on a NuGet package called **Azure.Messaging.ServiceBus**. You can install this package using the Package Manager Console in Visual Studio. To access the Package Manager Console, click **Tools > NuGet Package Manager > Package Manager Console.** Once in the Package Manager Console, run this command:

```
Install-Package Azure.Messaging.ServiceBus
```

Then, in any unit where you're writing Azure Service Bus code, be sure to include this line in your list of using statements:

```
using Azure.Messaging.ServiceBus;
```

Connect to the Service Bus Namespace

For both sending and receiving messages, you'll first need to instantiate a **ServiceBusClient** that manages the authentication and access to resources within a Service Bus Namespace. Here's an example of what creating a **ServiceBusClient** might look like:

```
string nameSpace =
  "TestBusNamespace.servicebus.windows.net";

DefaultAzureCredential creds = new();

ServiceBusClientOptions options = new()
{
    TransportType =
        ServiceBusTransportType.AmqpWebSockets
};
```

```
ServiceBusClient client =
   new(nameSpace, creds, options);
```

The use of **DefaultAzureCredentials** is great for a production environment with sophisticated Azure authentication flows in place. But for this little demonstration app, it's overkill. Instead, use a connection string. Like a SQL connection string, an Azure Service Bus connection string contains all the elements needed to identify and authenticate to an Azure Service Bus Namespace, including a key, so you'll want to take care to store these securely.

You can access a connection string for a queue via the Azure Portal by navigating to the Namespace and then selecting **Settings > Shared access policies > RootManagerSharedAccessKey** and then copying the **Primary Connection String** from the panel that appears on the right. You can also access the keys and connection strings via the Azure CLI with a command like this:

```
az servicebus namespace authorization-rule `
  keys list `
  --resource-group "Resource_group_1" `
  --namespace-name "TestBusNamespace " `
  --name "RootManageSharedAccessKey" `
  --query "primaryConnectionString" `
  --output tsv
```

Once you have your connection string (which will be quite long) you can save it to a configuration file or other secure storage and load it as needed. Now, the **ServiceBusClient** constructor will be quite a bit simpler:

```
string constr =
    ConfigurationManager.AppSettings["ConStr"];

ServiceBusClient client = new(constr);
```

That is much easier.

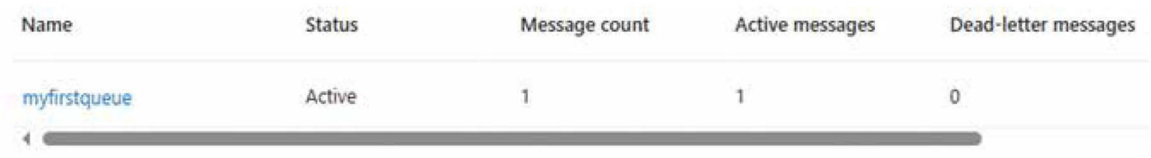
Send and Receive

The fundamental two actions when using Azure Service Bus are sending and receiving messages. Later in the article, I'll explore some of the complexities involved in constructing useful messages and building a production quality message receiver, but to start with, let's just send and receive a single message.

Send a Message

The basic pattern for sending a message is very simple: create a sender, create a message, send.

```
ServiceBusClient client = new(constr);
```



Name	Status	Message count	Active messages	Dead-letter messages
myfirstqueue	Active	1	1	0

Figure 4: The first message

```
string queue = "MyFirstQueue"

ServiceBusSender sender =
    client.CreateSender(queue);

ServiceBusMessage msg =
    new("This is my first message!");

await sender.SendMessageAsync(msg);
```

Once you run this code, the message should show up as an Active message in the Azure Portal, as shown in **Figure 4.**

Receive a Message

Receiving a message is also very simple: create a receiver, receive, complete.

```
ServiceBusClient client = new(constr);

string queue = "MyFirstQueue"

ServiceBusReceiver receiver =
    client.CreateReceiver(queue);

ServiceBusReceivedMessage msg =
   await receiver.ReceiveMessageAsync();

if (msg != null)
    await receiver.CompleteMessageAsync(msg);
```

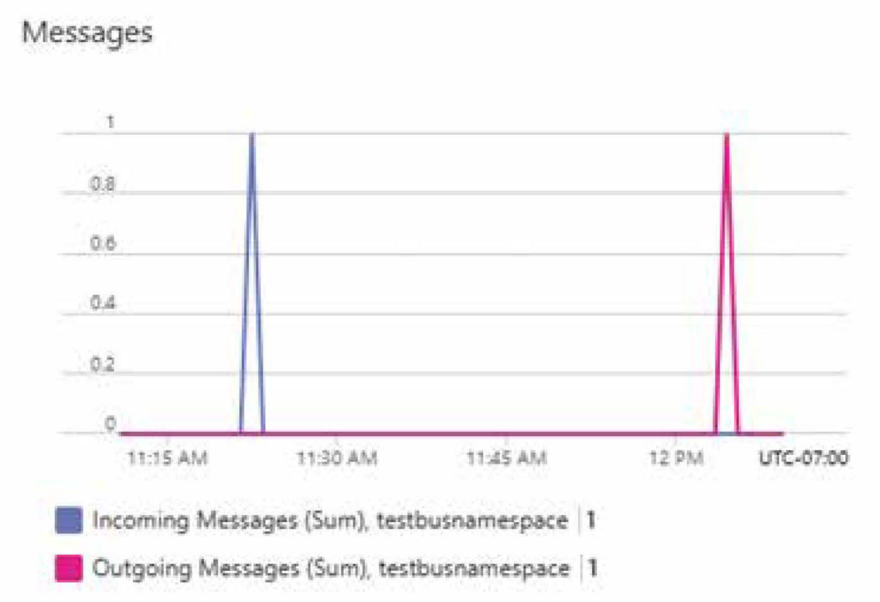


Figure 5: Send and receive events graphed on the Azure Portal

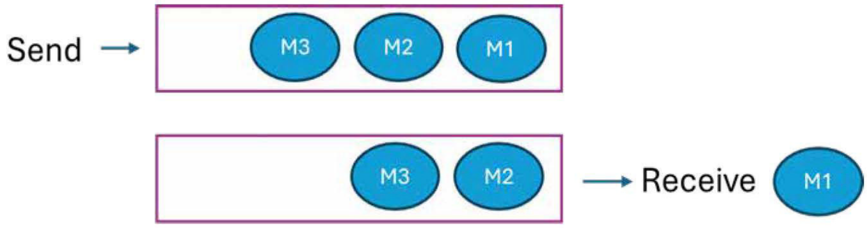


Figure 6: A queue is a FIFO data structure

Once you run this code, the Active message count in the queue returns to zero and you can see both the send and receive activity recorded in a handy graph, shown in **Figure 5.**

The one part of that code you might not have anticipated was the need to **Complete** the message after receiving it. You'll be looking at exactly what that's for later when I talk about **Peek-Locks**.

Message Delivery Order

What happens if there's more than one message in the queue? Which one does the receiver get? Remember, a queue is a first-in-first-out (FIFO) data structure. That means that if you send multiple messages to a queue, your receiver will receive them in the order they were sent, as shown in in **Figure 6.**

Azure Service bus does contain features like peeking and deferring that you can use to get around strict FIFO ordering. But as long as you stick with typical receiver patterns, it's easy to preserve strict FIFO if you want to.

Topics and Subscriptions

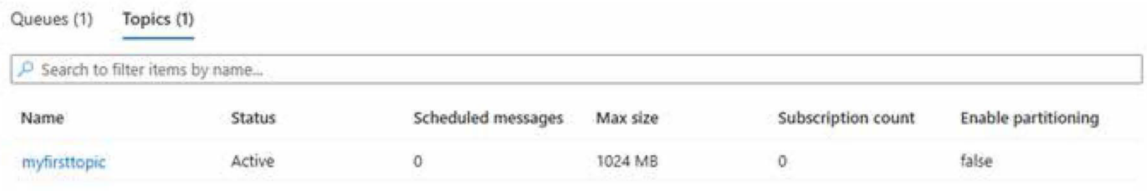
I'm sure you've heard the term "pub-sub" which is short for publish-subscribe or publisher-subscriber. The idea with a pub-sub architecture is that a publisher can publish a message once and multiple subscribers can then get the message. Azure Service bus supports this pattern very naturally with **Topics** and **Subscriptions**.

A Topic is an Azure Service Bus Queue that has been configured to allow multiple receivers, which are called Subscriptions. You create them the same way on the Azure Portal as you create Queues, except you use the **topic create** command:

```
az servicebus topic create `
   --resource-group Resource_group_1 `
   --namespace-name TestBusNamespace `
   --name MyFirstTopic
```

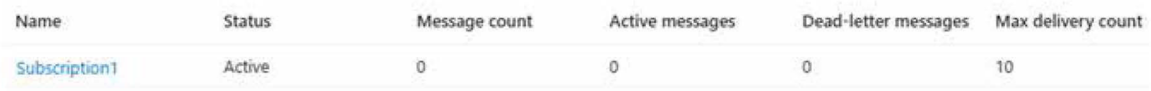
Once a Topic is created, it shows up in the Topics list of your Namespace, right next to the Queues, as shown in **Figure 7.**

Note that unlike a Queue, your Topic does not have Message count property, but it does have a Subscription count. More on that in a second.



Name	Status	Scheduled messages	Max size	Subscription count	Enable partitioning
myfirsttopic	Active	0	1024 MB	0	false

Figure 7: The first topic



Name	Status	Message count	Active messages	Dead-letter messages	Max delivery count
Subscription1	Active	0	0	0	10

Figure 8: The first Subscription

Once your **Topic** is set up, you can create multiple Subscriptions. Here's the CLI command to create one:

```
az servicebus topic subscription create `
    --resource-group Resource_group_1 `
    --namespace-name TestBusNamespace `
    --topic-name MyFirstTopic `
    --name Subscription1
```

Figure 8 shows the result in the Azure Portal.

Note that this **Subscription** looks just like a **Queue,** complete with **Message count** and all the other **Queue** properties. Why is that? When you send a message to a **Topic**, what really happens is a copy of that message gets sent to each **Subscription**, so each **Subscription** functions like an independent queue, as shown in **Figure 9.**

The **Topic** itself doesn't really hold any messages. It has no message counts but the Subscriptions do. This pattern continues as you start to receive messages. Let's say that the Sender posted a second message. Meanwhile, Subscription 1 receives the first message. The state would then look like **Figure 10.**

The key point here is that the three **Subscriptions** function as totally independent message queues. Receiving from one has no effect on the others, but sending to the **Topic** always sends to all **Subscriptions**.

> The code to send a message to a Topic is identical to the code for sending to a Queue except that you use the Topic Name.

The code to send a message to a **Topic** is identical to the code for sending to a **Queue** except that when you create the **ServiceBusSender**, you use the Topic Name:

```
string topic = "MyFirstTopic"

ServiceBusSender sender =
   client.CreateSender(topic);

ServiceBusMessage msg =
    new("Hello to all my subscribers!");
```

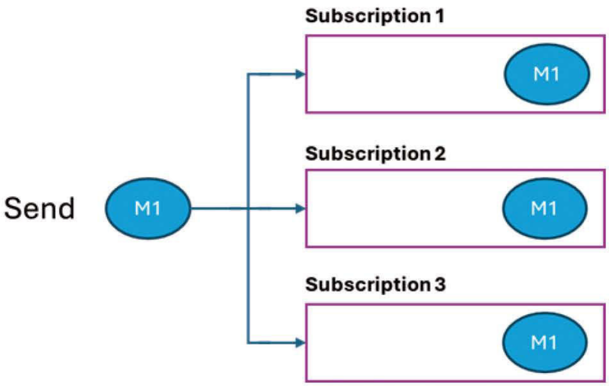


Figure 9: Sending to a Topic with three Subscriptions

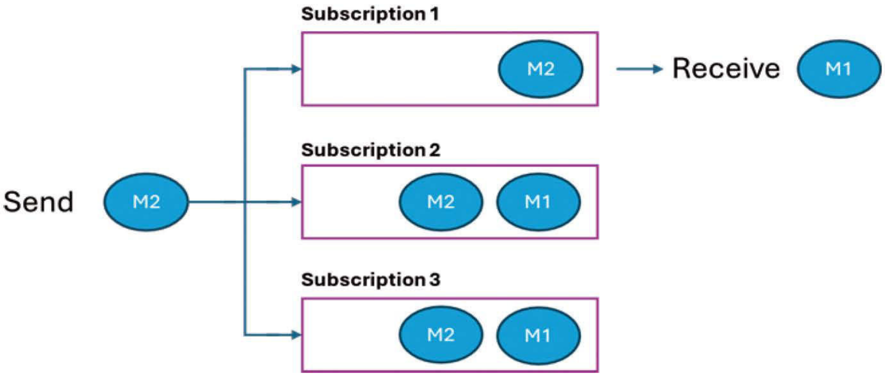


Figure 10: Receiving from a Subscription

```
await sender.SendMessageAsync(msg);
```

Similarly, the code to receive from a Subscription is identical to the code for receiving from a **Queue** except that when creating the **ServiceBusReceiver** you specify both the Topic Name and the Subscription name:

```
string topic = "MyFirstTopic"
string sub = "Subscription1"

ServiceBusReceiver receiver =
    client.CreateReceiver(topic, sub);

ServiceBusReceivedMessage msg =
   await receiver.ReceiveMessageAsync();
```

When receiving from a **Subscription**, you have all the same choices and options that I discussed for receiving from a **Queue**.

Message Payload

In this simple send and receive example, you used the message body, "This is my first message!" For a real application, you're going to want to send something a little more useful than that. Before you start building a message-based architecture, it's important to give some thought to the kinds of messages you plan to send and how that relates to the overall pattern and processing logic of your application.

Heavy vs. Light Messages

In the world of messaging-based applications, I generally see messages that fall into one of two categories that I call "heavy" and "light." Note that these are my terms and others may use different words for the same concepts (another phrasing I've seen is "stateful vs. stateless").

A **heavy** message is one where the body of the message contains **everything the receiver needs to process the message.** Consider a real-world example. Let's say you're a sales representative, and you're working on a bid for a customer. Your company has a strict approval process. Now imagine that it's 30 years ago, so to get your bid approved you need to print it out, put a sticky note on top saying "Please review", and place it in your manager's inbox, as shown in **Figure 11.**

In this case, the queue is the physical inbox and the message is the printed bid with your post it. This message contains everything the manager needs to review the bid. They don't have to ask you for a copy or pull it from a file cabinet. It's all right there. This is a "heavy" message.

Now fast-forward to the present day and you're using a modern CRM system to prepare your bids. As you work on the bids, the data is saved in a database of some sort. When you're ready for your manager to approve the bid, you click a "Submit for Approval" button, and a message gets posted to a collaboration channel saying **Bid 123 for Customer X is ready for your approval.** To review and approve the bid, the manager needs to **retrieve** the item from the database, as shown in **Figure 12.**

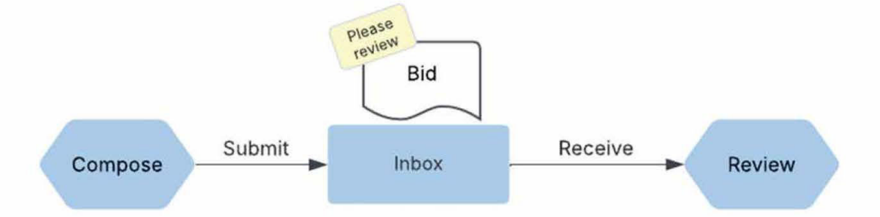


Figure 11: An example of a "heavy" message

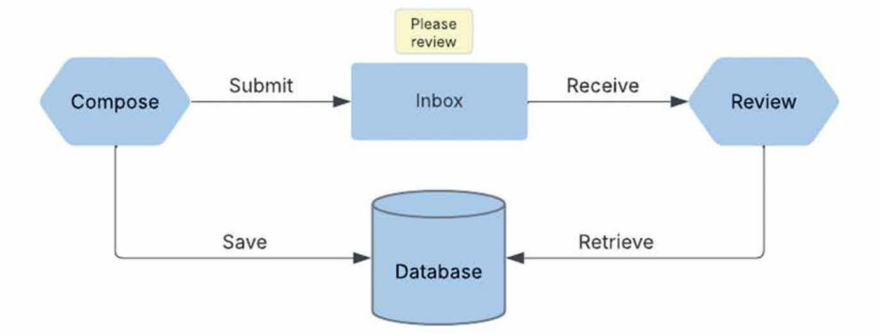


Figure 12: An example of a "light" message

In this case, the queue is the manager's collaboration feed and the message is a simple post notifying them that they have something to approve. The message is light. It does not contain everything needed for the review. Instead, it acts as a **signal** telling the manager that something happened.

This distinction between heavy and light messages has significant implications for your architecture. Both approaches have their pros and cons. Heavy messages are convenient. They free the application from having to grant every component access to a repository and it removes a shared dependency that can work against a truly decoupled architecture.

On the downside, heavy messages are stateful and if the real-world state of the data changes, they become stale, which can create data consistency issues, especially if you're not strict with message ordering. In addition, if you use heavy messages and these messages contain the only copy of the data, you need to be extra careful not to lose the messages because there's no way to recover the lost information. Imagine if your boss misplaced the only copy of that bid proposal!

Light messages are simple, easy to compose and support an event-based architecture where one component can signal another about an event and then the other can decide what action to take. The downside is that component systems may need to share some form of access to a repository that can tightly re-couple the systems.

I've used both approaches and sometimes have even mixed them in a single application. They key is to understand the difference and make an intentional choice.

JSON Serialized and Binary Content

A very common way to construct the body of an Azure Service Bus message is to use a simple model class and JSON serialization. This works for both light and heavy messages. Using the example of the bid sent for approval, you might begin with a simple model summarizing the bid:

```
public class Bid
{
    public int BidId {get; set;}
    public DateTime BidDate {get; set;}
    public string Customer {get; set;}
    public string Description {get; set;}
    public decimal Amount {get; set;}
}
```

Your code to construct the message would then look like this:

```
Bid bid = new()
{
    BidId = 1000,
    BidDate = DateTime.UtcNow,
    Customer = "ACME Widgets",
    Description = "Plastic Pellets",
    Amount = 25000M
};

ServiceBusMessage msg  =
    new(JsonSerializer.Serialize(bid));

// or

ServiceBusMessage msg = new() { Body = bid };
```

For a heavy message, you could simply use a much larger and more complex data model.

> A Service Bus Message Body is BinaryData, so you can send a complete document if you want. Just know your message size limitations that depend on subscription and protocol.

Interestingly, the Body property of a Service Bus Message is actually **BinaryData**. This means if you wanted to go with a really heavy message where a complete binary document is in the message, you could do something like this:

```
byte[] docBytes =
    await File.ReadAllBytesAsync(bidDocumentPath);

ServiceBusMessage msg = new()
{
    Subject = JsonSerializer.Serialize(bid),
    Body = new BinaryData(docBytes)
};
```

In this case, you're using the Subject of the message like the sticky note and the complete document is in the Body. Just remember that the Maximum Message Size in Azure depends on your subscription and protocol. On the Premium Tier using AMQP Web Sockets, messages can be up to 100 MB.

Correlation IDs

Before I move on from the subject of message payload, I want to talk about Correlation IDs. To me, Correlation IDs are one of the most important and useful features of any message-based or microservices architecture. In any decoupled system where there are multiple independent components that handle part of a process, there's often the need to trace the journey of a single request across the entire chain. Consider the example of an e-commerce application.

When you click "Submit Order" in your favorite shopping app, a whole series of actions happens. A credit card service processes your payment. An email service sends you a confirmation. A fulfillment service starts picking and packing your order. A shipping service ships your order. The email service notifies you of the shipment. The billing service may send you a statement. How do you map out the journey of a single order across all these systems, each of which may have its own repository and its own log mechanism?

The answer is to use Correlation IDs. A Correlation ID is an identifier (often a GUID), assigned by the first system in the chain, and then passed through in every subsequent message. This is so important that the Azure Service Bus Message class has an explicit **CorrelationId** property. To add a Correlation ID to the message is very easy:

```
ServiceBusMessage msg = new()
{
    Subject = JsonSerializer.Serialize(bid),
    Body = new BinaryData(docBytes),
    CorrelationId = Guid.NewGuid().ToString()
};
```

> Correlation IDs are one of the most important and useful features of any message-based or microservices architecture.

I strongly recommend that you use them.

Receiver Modes and Options

Whether you're using simple queues or topics and subscriptions and regardless of your message payload, there's quite a bit to understand and consider when it comes to receiving Azure Service Bus messages. You have a lot of options and behaviors to be aware of.

Receive Timeout

What happens if there are no more messages in the queue when you call **ReceiveAsync?** The answer is that your code will wait for sixty seconds (the default timeout) and then return **null**. You can customize the receive timeout by passing a TimeSpan to **ReceiveMessageAsync**. Regardless of the timeout you use, it's possible that **ReceiveMessageAsync** may return null**,** so before processing your message, you should always check that you did indeed get one:

```
TimeSpan timeout = TimeSpan.FromSeconds(5);

ServiceBusReceivedMessage msg =
   await receiver.ReceiveMessageAsync(timeout);

if (msg != null)
{
    // process the message...
}
```

Receive-and-Delete

Azure Bus supports two different receive modes: **Peek-Lock** and **Receive-and-Delete**. Understanding these receive modes is one of the most important concepts in the world of Azure Service Bus. Let's start with the simpler of the two modes, **Receive-and-Delete**.

With **Receive-and-Delete**, the act of receiving a message causes it to be **immediately and permanently deleted from the queue**. There's no need to do any additional processing, like completing the message. In fact, if you call **CompleteMessageAsync()** when using Receive-and-Delete mode, you'll get an exception.

> Azure Bus supports two different receive modes: Peek-Lock and Receive-and-Delete. Understanding these receive modes is one of the most important concepts in the world of Azure Service Bus.

The advantage of this approach is that the behavior is simple and easy to understand. The downside is that you can't take advantage of some of the sophisticated retry behaviors outlined below. You get one shot at the message and then it's deleted.

It's important to note that **Receive-and-Delete** is **not** the default receive mode. To use it, you need to explicitly configure your receiver as shown:

```
ServiceBusReceiverOptions options = new()
{
    ReceiveMode =
        ServiceBusReceiveMode.ReceiveAndDelete
};

ServiceBusReceiver receiver =
    client.CreateReceiver(queue, options);

ServiceBusReceivedMessage msg =
    await receiver.ReceiveMessageAsync(timeout);
```

Peek-Lock

With **Peek-Lock**, the act of receiving a message is more complicated, and potentially more confusing. When you receive the message, the server sets up a peek-lock on your message. This means that the message is marked as locked and no other receiver can receive it while it's locked. Then the server starts a timer. Your client code needs to take some sort of completion action before that timer expires. You have four options, as shown in **Table 1.**

Choosing the correct completion option is critical. If you processed the message successfully, you want to **Complete** it. But what if something goes wrong? This gets tricky and your choice depends a lot on the nature of the problem and the logic of your application. If you're dealing with a "poison message" (i.e., one that will **never** be able to be processed) it's best to just **Dead Letter** it right away, otherwise you risk clogging up your queue with bad messages. However, if the message failed but you might be able to recover and process it on a retry, then maybe **Abandon** is the right move. I'll return to topics related to poison messages and other issues at the end of the article.

Peek-Lock Expiration

What happens if your peek-lock expires? On the server side, if a peek-lock expires, the server treats it like an **Abandon**. This means the message is unlocked and the retry counter is incremented. On the client side, if your peek-lock expires as soon as you call one of the completion methods, you'll receive an exception.

There are a couple of techniques available to you to help manage Peek-Lock expirations. First, you can configure the Message lock duration. On the Azure Portal, locate your queue and click to see the details. Under settings, you can access the Message lock duration, as shown in **Figure 13.**

Click Change to adjust the duration from five minutes to five seconds. This approach can be helpful if, on a routine basis, message processing takes a bit longer than one minute. By simply expanding the lock, you can save yourself from experiencing unwanted timeouts.

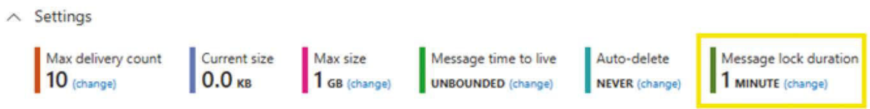
The other approach you can take is to renew the lock using **RenewMessageLockAsync**(). Conceptually, this is simple, but implementing it is a little tricky because your code needs to somehow keep track of how much time has expired while you're processing your batch of messages. This involves some parallel processing code, which is beyond the scope of this article.

Receiver Patterns

So far, you've looked at code that receives a single message. A real application is going to receive and process

Option	Method	Description
Complete	CompleteMessageAsync(msg);	The message is removed from the queue. This is used for normal, successful message processing.
Dead Letter	DeadLetterMessageAsync(msg);	The message is removed from the queue and placed in a special Dead Letter Queue. This is generally used when a message is not processable. It's possible to "replay" messages in the Dead Letter queue using the Azure Portal or CLI.
Abandon	AbandonMessageAsync(msg);	The message is unlocked and its retry counter is incremented. This is used when the message might be able to be processed in the future, so you want to put it back in the queue and retry later. The number of allowed retries is configurable. When a message hits this maximum number of allowed deliveries, the next Abandon will send it to the Dead Letter Queue instead.
Defer	DeferMessageAsync(msg);	The message technically stays in the main queue but is marked as deferred. In order to receive this message again in the future, you need to save its SequenceNumber and receive it using the special ReceiveDeferredMessageAsync method. This approach creates complex message ordering and access issues and should be used with caution.

Table 1: Four options for receiving a message



Settings

Max delivery count 10 (change)

Current size 0.0 KB

Max size 1 GB (change)

Message time to live UNBOUNDED (change)

Auto-delete NEVER (change)

Message lock duration 1 MINUTE (change)

Figure 13: Queue settings highlighting message lock duration

multiple messages over a long period of time. There are several ways you can approach this.

A Message Pump

A message pump is a long-running loop that processes messages as they come in, like this:

```
while (true) // long running loop
{
    msg = await
        receiver.ReceiveMessageAsync(timeout);
    if (msg != null)
    {
        // process the message

        await receiver.CompleteMessageAsync(msg);
    }
}
```

Please note that this loop is missing some very important parts. For production code, you need to add exception handling and a way to gracefully exit the loop with something like a **CancellationToken**. But this gives you the basic idea of how you could start writing a message-pump based Service Bus Receiver. The Visual Studio Sample Project associated with this article (and which you can download from www.CODEMag.com) has a more fully realized message pump for you to try out.

Batch Processing

All of the receiver code up to this point has operated one message at a time, but it's quite possible to create a Service Bus Receiver that works in a more batch-based fashion by receiving a set of messages:

```
IReadOnlyList<ServiceBusReceivedMessage>
    messages =
    await receiver.ReceiveMessagesAsync(10,
   timeout);

foreach (ServiceBusReceivedMessage msg
    in messages)
{
    // process the message

    await receiver.CompleteMessageAsync(msg);
}
```

In this example, you passed the number 10 to **ReceiveMessagesAsync**. This is the maximum number of messages you want returned in the batch. If there were 25 messages in the queue, you'd get the first 10. If there were only seven messages in the queue, you'd get all seven. Interestingly, you can combine the Message Pump pattern with Batch Processing and write a long-running loop that pulls batches of messages instead of single messages.

One thing to be aware of with this kind of batch processing technique: When you receive a batch of messages, the **peek-lock timer starts for all of the messages in the batch**. This can lead to unexpected lock timeouts if you're not careful. Let's take a moment to think this through and see how you might encounter unexpected lock expirations.

> When you receive a batch of messages, the peek-lock timer starts for all of the messages in the batch. This can lead to unexpected lock timeouts if you aren't careful.

Let's say that your lock timeout has the default of one minute. Then, let's say you know that it takes about 10 seconds to process each message. Doing the math, you calculate that you can realistically pull five messages at a time because 5 * 10 = 50 seconds, which is comfortably under your timeout of 60 seconds.

Now, imagine that, for some reason, messages are moving slowly. Maybe there's a dependent service that's lagging, so now, your messages are taking 20 seconds each, rather than 10. Here's what happens:

- Message 1: Total elapsed time 20 seconds,
- Message 2: Total elapsed time 40 seconds
- Message 3: Total elapsed time 60 seconds
- Message 4: Total elapsed time: **Error–Peek-Lock Timeout**
- Message 5: Total elapsed time: **Error–Peek-Lock Timeout**

You have several ways to manage this situation, such as lengthening the peek-lock timeout, reducing your batch size, and tracking elapsed time, so you can renew locks if needed. Just be aware of how this works. If you don't plan for it, it can be a major gotcha.

Event-Based Message Receiver

The Azure Service Bus namespace contains a useful class called the **ServiceBusProcessor**. This class implements a message pump for you and provides an elegant, event-based programming model. Under the hood, **ServiceBusProcessor** is, in fact, a wrapper around one or more **ServiceBusReceivers**. Many of the Azure Service Bus quick-starts and tutorials you'll find start with the ServiceBusProcessor because it's so easy to use:

```
ServiceBusProcessor processor =
    client.CreateProcessor(queue);

processor.ProcessMessageAsync += OnMessageAsync;
processor.ProcessErrorAsync += OnErrorAsync;
```

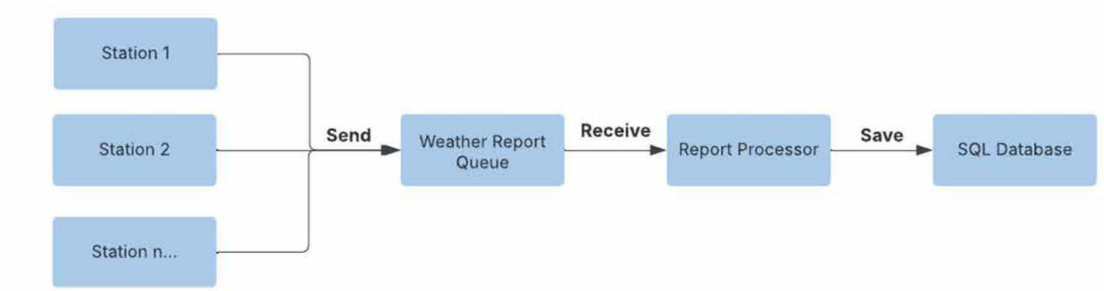


Figure 14: A queue-based weather reporting app

```
await processor.StartProcessingAsync();
```

In this code, you create a ServiceBusProcessor, wire up two event handlers, and then start the processor. Here are what the event handlers look like:

```
async Task OnMessageAsync(
    ProcessMessageEventArgs args)
{
    // process message here
}

async Task OnErrorAsync(
    ProcessErrorEventArgs args)
{
    // process error here
}
```

By default, the **ServiceBusProcessor** is using Peek-Lock as the receive mode and it's pulling one message at a time. It's also using a feature called **autocomplete,** which means that at the end of your **ProcessMessageAsync** handler, the message is automatically completed for you. You don't need to call **CompleteMessageAsync**. Likewise, **ProcessErrorAsync** handler automatically calls **AbandonMessageAsync** for you.

> The ServiceBusProcessor class implements a message pump for you and provides an elegant, event-based programming model.

You can control all these behaviors by using **ServiceBusProcessorOptions**. For example, let's say you wanted to use Retrieve-and-Delete with a batch-based receiver that can pull ten messages at a time. Your code would look list this:

```
ServiceBusProcessorOptions options = new()
{
    ReceiveMode =
        ServiceBusReceiveMode.ReceiveAndDelete,
    PrefetchCount = 10
};

ServiceBusProcessor processor =
    client.CreateProcessor(queue, options);
```

The Visual Studio sample project associated with this article on www.codemag.com also has a ServiceBusProcessor for you to try.

Errors and Fault Tolerance

If you stopped reading this article right now, you'd likely have enough to go build a workable, messaging-based application using Azure Service Bus. But before I wrap up, I do want to talk about some error conditions that you need to anticipate and have a plan for so that your Azure Service Bus Solution will be robust and fault tolerant.

To facilitate the conversation about errors, it's helpful to have a more specific application example in mind. Imagine that you have a Weather Data collection application. There are weather stations all over that periodically send weather reports to an input queue. From there, a report processor receives the messages, validates them, and writes them to a SQL Server database, as shown in **Figure 14.**

With this application in mind, let's look at the two most important kinds of error conditions to plan for: poison messages and transient error conditions.

Poison Messages

A poison message is a message that can never be processed by your application. No matter how many times you retry it, it will never succeed. Imagine that the Weather Report application uses a **ServiceBusProcessor** and the **ProcessMessageAsync** event handler looks like this:

```
async Task OnMessageReceived(
    ProcessMessageEventArgs args)
{
    string json = args.Message.Body.ToString();

    WeatherData report JsonSerializer.
        Deserialize<WeatherData>(json); // poison?

    // more message processing code...
}
```

What happens if the message body isn't valid JSON? When you attempt to deserialize it, the call to **Deserialize** throws a **System.Text.Json.JsonException.** This is a poison message. No matter how many times you try to deserialize it, it will never work.

If you leave the code above as is (without any exception handling), that exception will cause your **ProcessErrorAsync** event handler to fire. Let's say this is your handler:

```
async Task OnMessagError(
    ProcessErrorEventArgs args)
{
    await LogError(args.Exception.Message);
}
```

What's the end result? If you're using **Peek-Locks** and the default **Max Delivery Count** of 10, this poison message is going to retry 10 times and then, after the tenth try, it ends up in the DeadLetterQueue. This is not a great behavior. Now imagine that instead of one bad message, you have thousands. Maybe it's a bad actor intentionally sending poison messages. Your Service Bus Processor will dutifully retry them all (10 times by default) and then they will all end up in the Dead Letter Queue. In the best case, your app churns and it costs you money on your Azure subscription. In the worst case, valid requests start getting choked off.

The solution is to identify poison messages on the first attempt to process them and Dead Letter them immediately. One way to do this in the specific example is with a typed exception handler like this:

```
try
{
    // all the same code to process a message
}
catch (JsonException) // poison message
{
    await
      args.DeadLetterMessageAsync(args.Message);
}
```

Now you can eliminate this kind of Poison Message more efficiently.

Transient Error Conditions

Let's consider a different kind of error. Let's say that after deserializing the weather report, you need to write to SQL. That code might look like this:

```
WeatherData report JsonSerializer.
    Deserialize<WeatherData>(json);

using SqlConnection con = new(sqlConStr);

await con.OpenAsync(); // what if SQL is down?

await WriteToSql(con, report);
```

What happens if the SQL Server is unavailable for a while? The call to **Connection.OpenAsync** is going to throw an exception, and this means that during the SQL outage, every single message that comes through is going to fail and retry 10 times and then end up in the Dead Letter Queue. In practical terms, you might suddenly find thousands or tens of thousands of messages piled up in the Dead Letter Queue just because you had a SQL Server service interruption. This is a big mess. Your server was churning for hours doing nothing but logging errors, and now you have a pile of viable messages to somehow replay.

The trick to handling these kinds of transient error conditions is to implement a "stand-off and try later" approach. Once you know that SQL Server is unavailable, there's no point processing any more messages. You want to pause your service bus processor and only re-enable it once you have verified SQL availability:

```
try
{
    // all the same code to process a message
}
catch (SqlException) // stand off and try later
{
    await args.AbandonMessageAsync(args.Message);
    await PauseTheProcessor();
}
```

The **PauseTheProcessor** might look something like this:

```
async Task PauseTheProcessor()
{
    await processor.StopProcessingAsync();
    while (true)
    {
        if (await TestTheSqlConnection())
        {
            await processor.StartProcessingAsync();
            break;
        }
        await Task.Delay(60000);
    }
}
```

I should note that this technique applies not only to a SQL Server connection but to **any dependencies** in your code. Maybe you're pulling data from one or more APIs. Maybe you need to send messages to other queues. Any one of these external dependencies is a possible source of transient error conditions. You need to try to account for them all.

Conclusion

At a high level, a message-based architecture is easy to conceptualize, and Azure Service Bus provides tools that make them easy to build. Below the surface, you have some important choices to make and situations to consider if you want a robust message-based application to really do what you want:

- Use simple **Queues** or **Topics and Subscriptions.**
- Use **heavy** (stateful) or **light** (stateless) messages.
- Use **Peek-Lock** or **Receive-and-Delete**.
- Use the **ServiceBusProcessor** or roll your own message pump.
- Pull **one message** at a time or use **batches**.
- Enforce strict **FIFO** ordering or allow techniques that can end up with messages out of order.
- **Abandon** and retry failed messages or **Dead Letter** them.

In summary, you have a lot to think about when building a Service Bus architecture: I find that it usually takes some careful thinking and a bit of trial and error to find which pattern and combination works best. My hope is that you now have enough information to get started and make solid choices with Azure Service Bus.

Matthew Hess

SPONSORED SIDEBAR:

Ready to Modernize a Legacy App?

Need advice on migrating yesterday's **legacy applications** to today's modern platforms?

Take advantage of **CODE Consulting's** years of experience and contact us today to schedule a **FREE** consulting call to discuss your options.

No strings. No commitment.

For more information: www.codemag.com/consulting or email us at info@codemag.com

Securing ASP.NET Core Blazor Applications

Joydip Kanjilal
joydipkanjilal@yahoo.com

Joydip Kanjilal is an MVP (2007-2012), software architect, author, and speaker with more than 20 years of experience. He has more than 16 years of experience in Microsoft .NET and its related technologies. Joydip has authored eight books, more than 500 articles, and has reviewed more than a dozen books.

Blazor is a modern web framework from Microsoft used for building interactive web applications using C# and .NET. It's based on a flexible, modular component model well suited for building applications with rich, interactive web user interfaces. Blazor leverages the authentication mechanism of ASP.NET Core to establish the identity of a user. You've had a glimpse at the features and benefits of Blazor applications and how you can implement them in ASP.NET Core in a previous article (here: https://www.codemag.com/Article/2503041/Building-Modern-Web-Applications-Using-Blazor-ASP.NET-Core). The article you're reading now takes a deep dive into how you can secure Blazor applications, with relevant code examples to demonstrate the concepts and the best practices.

If you're to work with the code examples discussed in this article, you need the following installed in your system:

- Visual Studio 2022
- .NET 9.0
- ASP.NET 9.0 Runtime

If you don't already have Visual Studio 2022 installed on your computer, you can download it from here: https://visualstudio.microsoft.com/downloads/.

At the end of this journey, you'll be able to build high-performance, scalable, and secure Blazor applications in ASP.NET Core 9 and Visual Studio 2022.

Application Security and Its Importance

Securing your applications has become more critical recently because of the surge in security breaches, malware, and other cybersecurity threats. Application security aims to determine and resolve security threats and vulnerabilities in an application before attackers exploit them. Your applications must be secure to avert exposure to confidential information, prevent unauthorized information access through hacking, reduce data breaches, bolster customer trust, and comply with legal requirements and guidelines.

Securing Blazor Applications

When working with Blazor applications, implementing robust security measures is an imperative because of:

- Protecting sensitive data from being compromised
- Thwarting unauthorized access to your application
- Protecting against cross-site scripting (XSS) attacks

You can secure your Blazor applications in several ways:

- Using HTTPS
- Using input validation
- Using anti-forgery tokens
- Leveraging ASP.NET Core security framework
- Using token-based authentication

In the sections that follow, I'll examine how to take advantage of these security strategies to protect your Blazor applications.

Secure a Blazor Server Application

A Blazor Server application is hosted on the server side—its components run on the server inside an ASP.NET Core application. The user interface is rendered on the server and sent to the web browser. It uses a SignalR connection and the WebSockets protocol to perform UI updates and handle events.

Anatomy of a Blazor Server Application

The solution structure of a typical Blazor Server application is comprised of several folders and files. The following is the list of folders in your Blazor Server application:

- **Areas:** Splits your Blazor Server application into a collection of functional groups. Each of these functional groups contains a set of Razor pages, controllers, views, and models.
- **Data:** The services and models used to implement the business logic of your application.
- **Pages:** The visual elements of your Blazor Server application, such as the layout template, main page, and the starting point of the application.
- **Shared:** The types that are shared across all components of your Blazor Server application, such as MainLayout.*, NavMenu.*, and so on.
- **wwwroot:** The static resources used by your Blazor Server application such as, CSS files and favicon.

Here is the list of the files used in your Blazor Server application and their purpose:

- **Program.cs:** Contains the startup code of your Blazor Server application
- **App.razor:** Represents the main Razor component of your Blazor Server application
- **_Imports.razor:** Contains the necessary code to import the required namespaces used in your Blazor Server application
- **appsettings:** Contains configuration data used in your Blazor Server application

Create a New Blazor Server Application in .NET 9 and Visual Studio 2022

You can create a project in Visual Studio 2022 in several ways, such as from the Visual Studio 2022 Developer Command Prompt or by launching the Visual Studio 2022 IDE. When you launch Visual Studio 2022, you'll see the Start window. You can choose **Continue without code** to launch the main screen of the Visual Studio 2022 IDE.

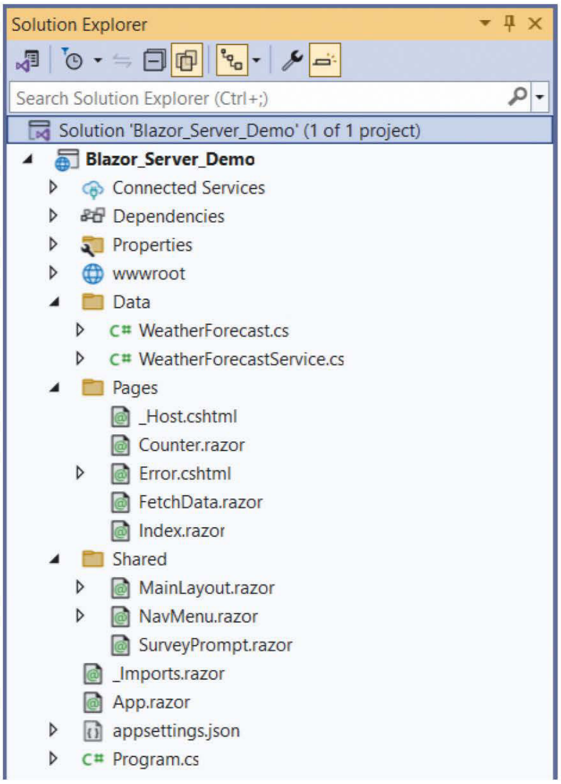
Now that you know the basics, let's start setting up the project. To create a new ASP.NET Core 8 Project in Visual Studio 2022:

1. Start the Visual Studio 2022 IDE.
2. In the **Create a new project** window, select **Blazor Server App** and click Next to move on.
3. Specify the project name as **Blazor_Server_Demo** and the path where it should be created in the **Configure your new project** window.
4. In the next screen, specify the target framework and authentication type as well. Ensure that the **Configure for HTTPS**, and **Do not use top-level statements** checkboxes are unchecked because you won't use any of these in this example.
5. If you want the solution file and project to be created in the same directory, you can optionally check the Place solution and project in the same directory checkbox. Click Next to move on.
6. Click Create to complete the process.

A new Blazor Server application project is created. **Figure 1** shows the default solution structure of the Blazor Server App you just created.

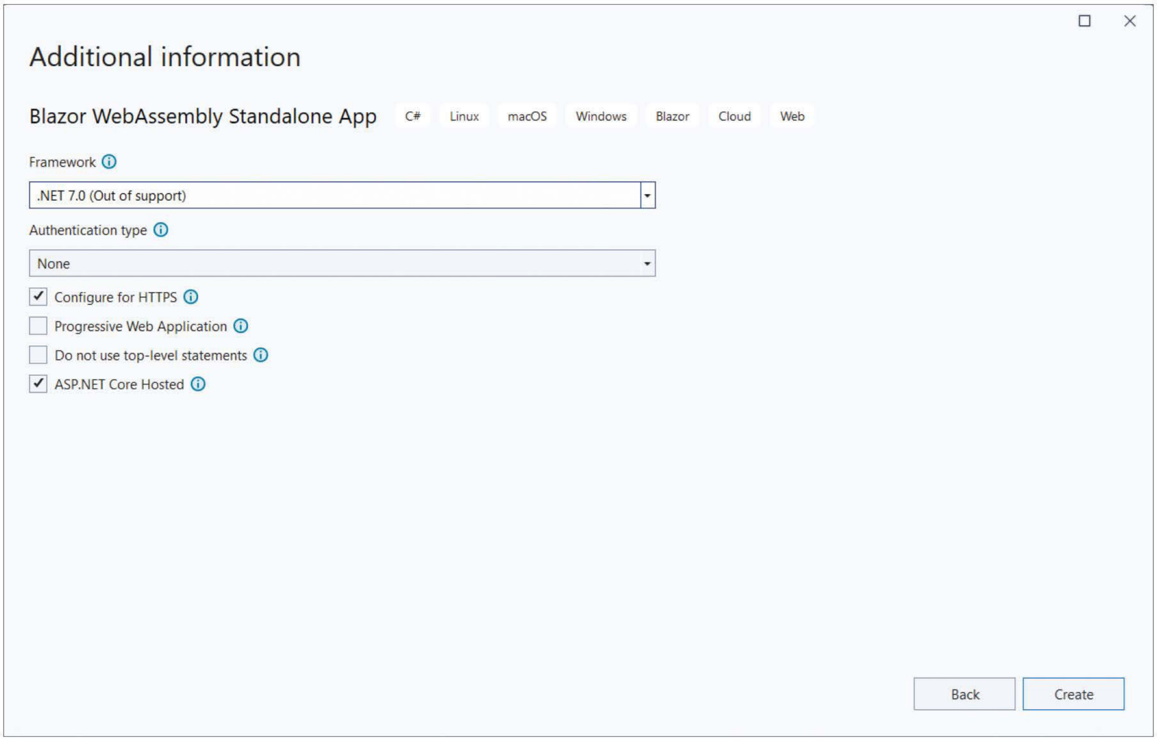
You can implement authentication in your Blazor application using the built-in authentication provided by Microsoft for Blazor applications. To use the built-in authentication mechanism, follow the same sequence of steps outlined in the preceding section to create a Blazor Server application. However, you should select the Authentication type you'd like to use in the application. In this example, select the Authentication type as Individual Accounts in the Additional information window, as shown in **Figure 2**.

Finally, click Create to complete the process. A new Blazor Server application is created with authentication enabled. When you execute the application, you'll see the Register and Login buttons in the header, as shown in **Figure 3**.



- Solution 'Blazor_Server_Demo' (1 of 1 project)
  - **Blazor_Server_Demo**
    - Connected Services
    - Dependencies
    - Properties
    - wwwroot
    - Data
      - WeatherForecast.cs
      - WeatherForecastService.cs
    - Pages
      - _Host.cshtml
      - Counter.razor
      - Error.cshtml
      - FetchData.razor
      - Index.razor
    - Shared
      - MainLayout.razor
      - NavMenu.razor
      - SurveyPrompt.razor
    - _Imports.razor
    - App.razor
    - appsettings.json
    - Program.cs

Figure 1: The Solution Explorer of a typical Blazor Server application



Framework: .NET 7.0 (Out of support)

Authentication type: None

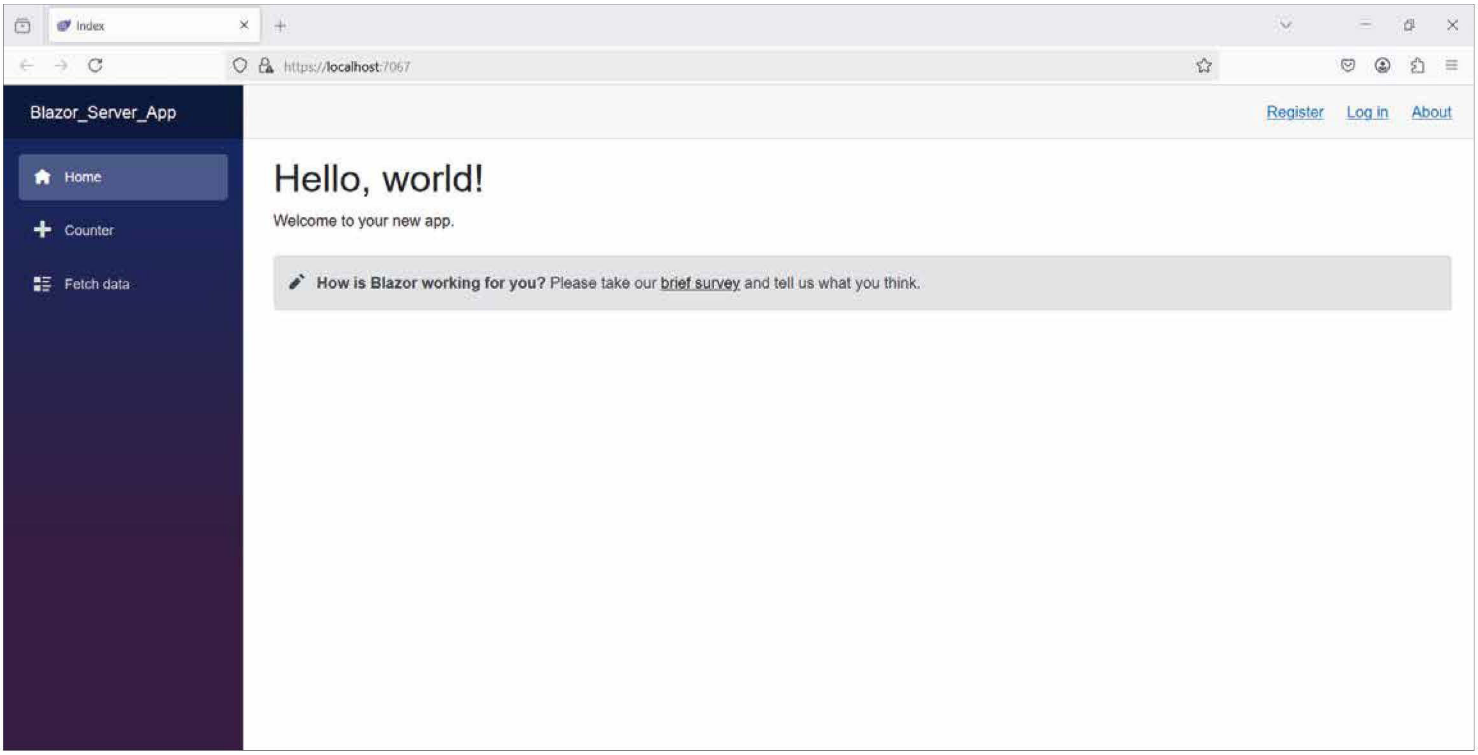
☑ Configure for HTTPS
☐ Progressive Web Application
☐ Do not use top-level statements
☑ ASP.NET Core Hosted

Figure 2: Specify additional information while creating a new Blazor Server application

As shown in **Figure 3**, there are three hyperlink buttons displayed at the top of the webpage. Because no application user has been created yet, you can click on the Register button to register a new user in the application. Now, provide the necessary authentication details for the new user referring to **Figure 4**.

Because no user has been registered yet, you'll be presented with the screen in **Figure 5** prompting you to execute migrations.

Next, click on the Apply Migrations button to run migrations and set up the database. Once the migrations have



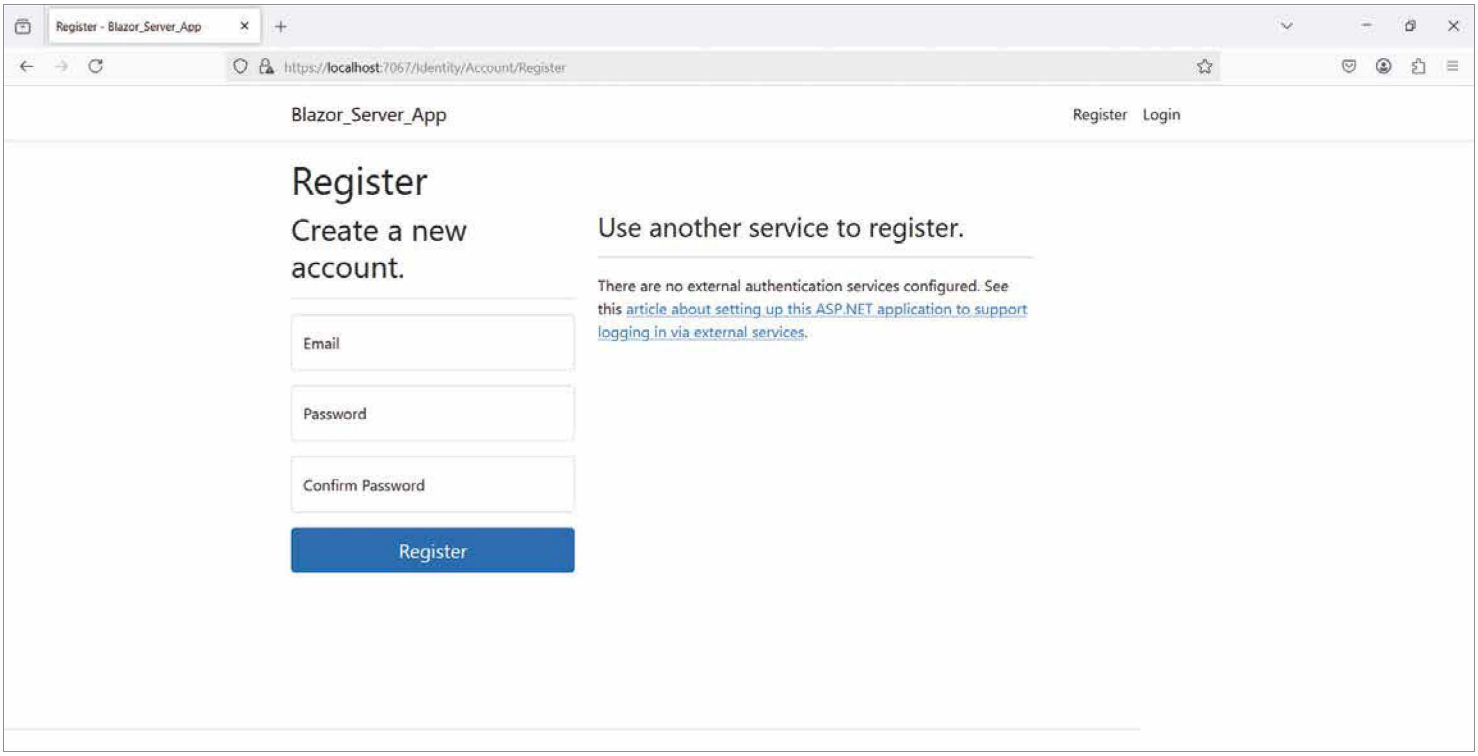
Index
https://localhost:7067
Blazor_Server_App
Register
Log in
About
Home
Counter
Fetch data

Hello, world!

Welcome to your new app.

How is Blazor working for you? Please take our brief survey and tell us what you think.

Figure 3: The Blazor Server application (with authentication support) in execution



Register - Blazor_Server_App
https://localhost:7067/Identity/Account/Register
Blazor_Server_App
Register
Login
Email
Password
Confirm Password
Register

Register

Create a new account.

Use another service to register.

There are no external authentication services configured. See this article about setting up this ASP.NET application to support logging in via external services.

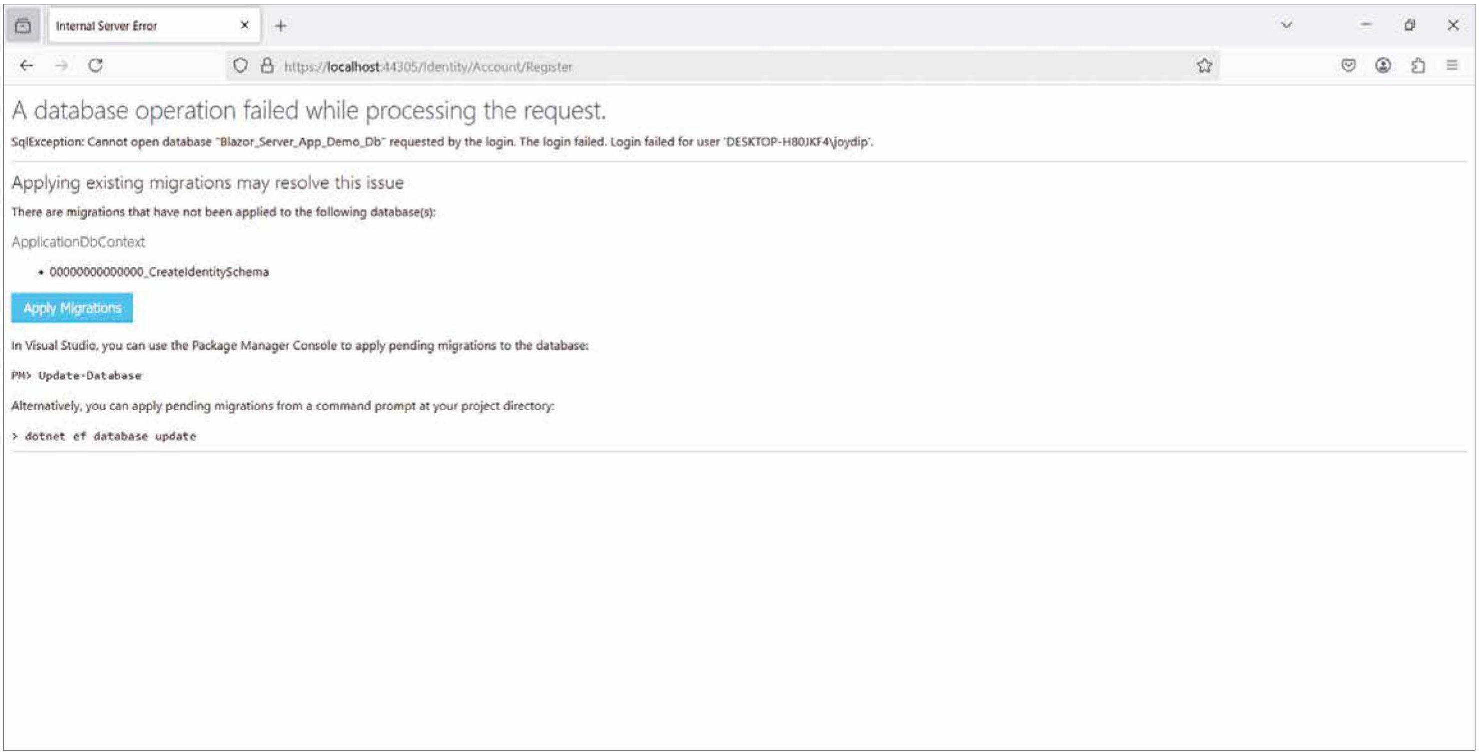
Figure 4: Register a new user account.

been executed, you'll be presented with the screen shown in **Figure 6**.

Once you refresh the screen, you'll be presented with a screen that prompts you to confirm your account, as shown in **Figure 7**.

Once you've confirmed your account, you'll be presented with the screen shown in **Figure 8** that says registration is confirmed for the user.

Finally, you can click on the Login button to invoke the Login screen where you can enter the user's credentials to



A database operation failed while processing the request.

SqlException: Cannot open database "Blazor_Server_App_Demo_Db" requested by the login. The login failed. Login failed for user 'DESKTOP-H80JKF4\joydip'.

Applying existing migrations may resolve this issue

There are migrations that have not been applied to the following database(s):

ApplicationDbContext

- 00000000000000_CreateIdentitySchema

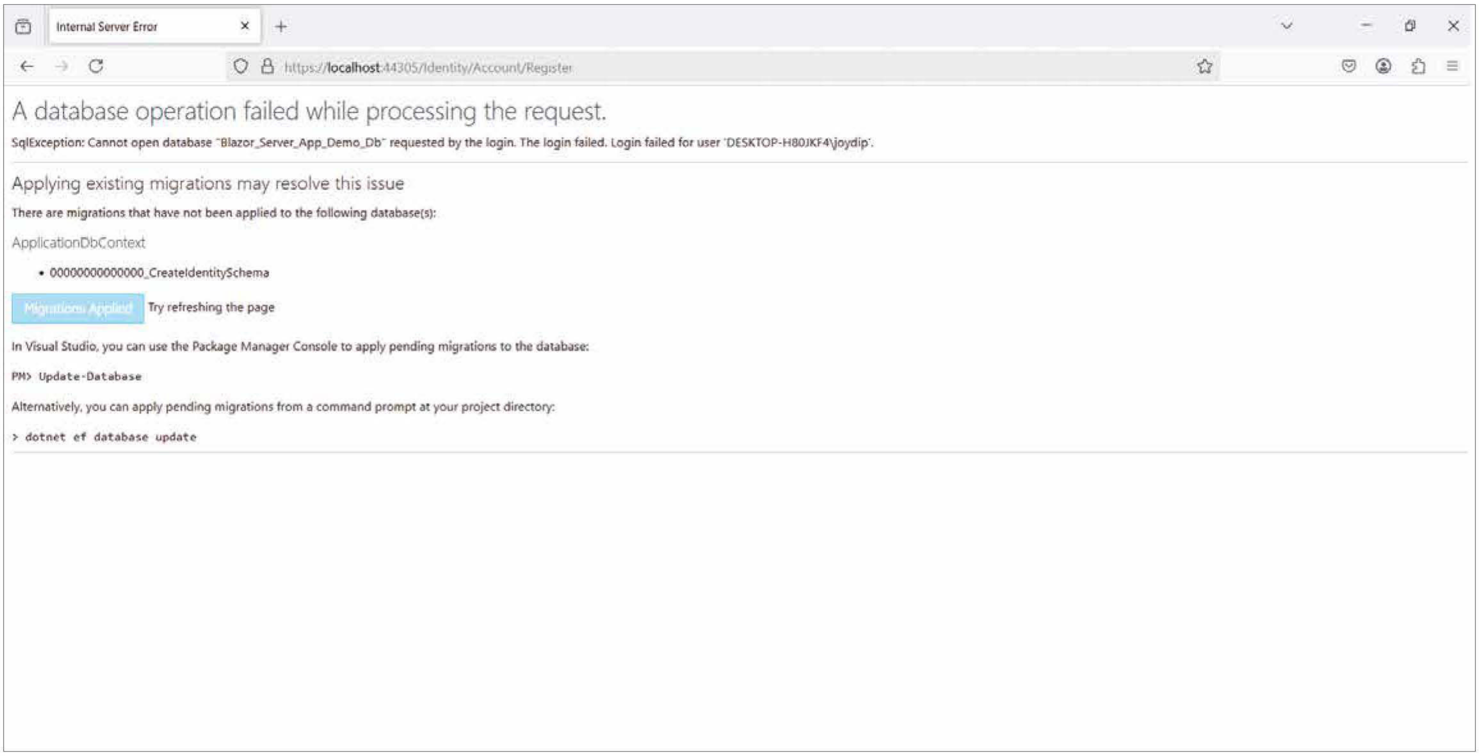
In Visual Studio, you can use the Package Manager Console to apply pending migrations to the database:

```
PM> Update-Database
```

Alternatively, you can apply pending migrations from a command prompt at your project directory:

```
> dotnet ef database update
```

Figure 5: You're prompted to apply migrations to create and initialize the database.



A database operation failed while processing the request.

SqlException: Cannot open database "Blazor_Server_App_Demo_Db" requested by the login. The login failed. Login failed for user 'DESKTOP-H80JKF4\joydip'.

Applying existing migrations may resolve this issue

There are migrations that have not been applied to the following database(s):

ApplicationDbContext

- 00000000000000_CreateIdentitySchema

Try refreshing the page

In Visual Studio, you can use the Package Manager Console to apply pending migrations to the database:

```
PM> Update-Database
```

Alternatively, you can apply pending migrations from a command prompt at your project directory:

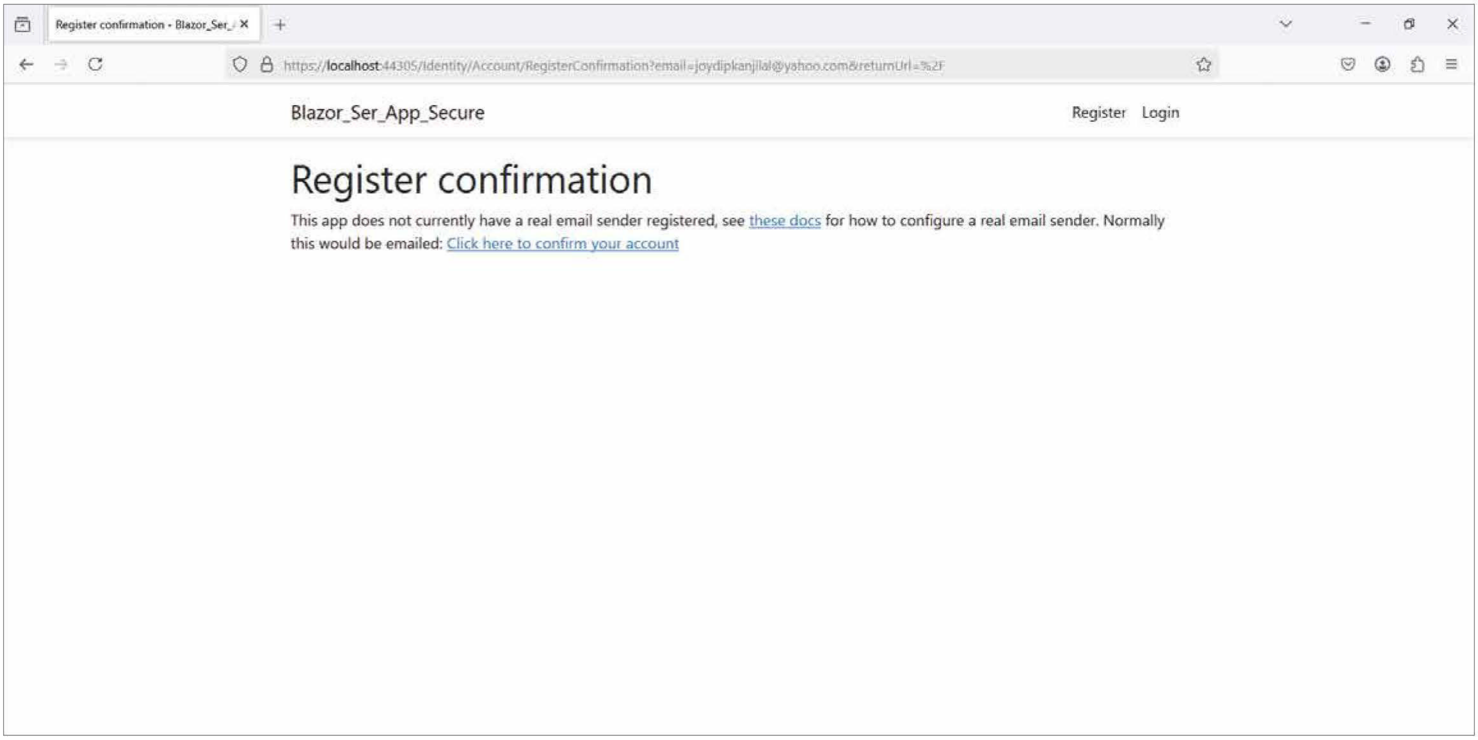
```
> dotnet ef database update
```

Figure 6: Migrations have been applied successfully.

log in and access the application, as shown in **Figure 9**. You can then use the Login screen to enter user credentials to log in to the application.

If the entered credentials are valid, you'll see the screen shown in **Figure 10** showing that the user has been authenticated and logged in to the application.

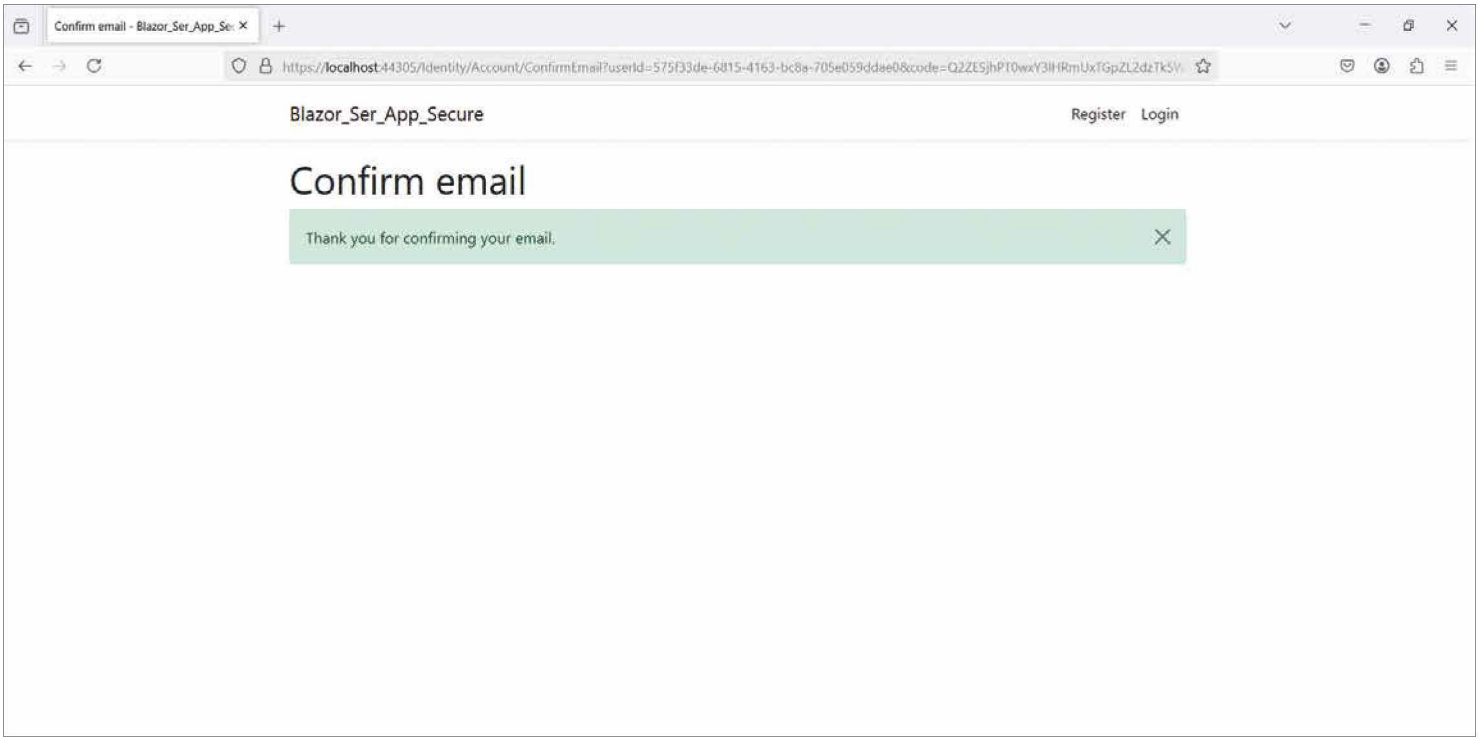
If you already have a Blazor application with you, you can't follow this approach to implement authentication in your Blazor applications. It's preferable to implement an out-of-the-box authentication to overcome the limitations of this approach. In the next section, you'll examine how you can take advantage of custom authentication to secure the application.


Register confirmation - Blazor_Ser_
https://localhost:44305/Identity/Account/RegisterConfirmation?email=joydipkanjilal@yahoo.com&returnUrl=%2F
Blazor_Ser_App_Secure
Register
Login

Register confirmation

This app does not currently have a real email sender registered, see these docs for how to configure a real email sender. Normally this would be emailed: Click here to confirm your account

Figure 7: User account registration confirmation


Confirm email - Blazor_Ser_App_Se
https://localhost:44305/Identity/Account/ConfirmEmail?userId=575f33de-6815-4163-bc8a-705e059ddae0&code=Q2ZESjhPT0wxY3lHRmUxTGpZL2dzTk5V
Blazor_Ser_App_Secure
Register
Login

Confirm email

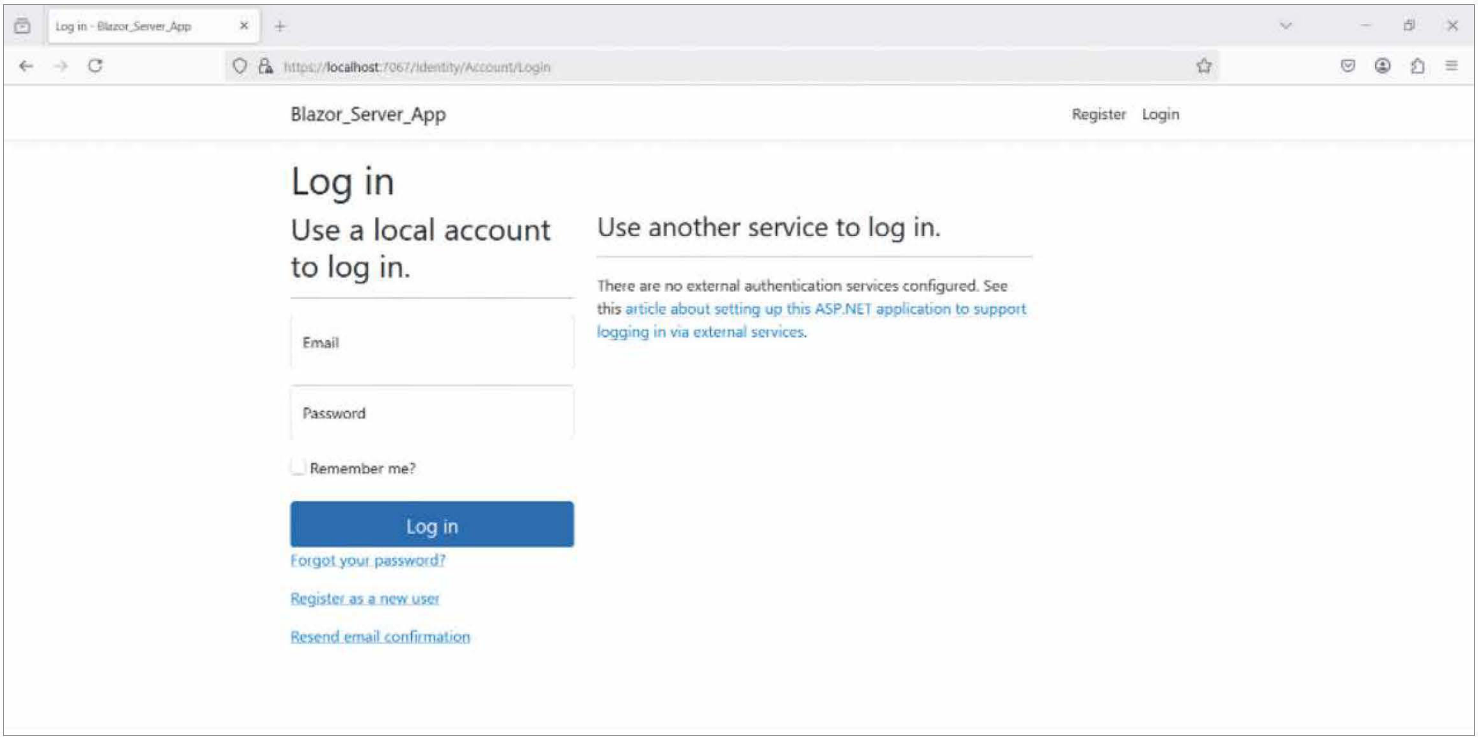
Thank you for confirming your email.

Figure 8: Registration confirmed

The AuthorizeView Component

An AuthorizeView component named LoginDisplay.razor will also be created in the Shared folder having the following code:

```
<AuthorizeView>
    <Authorized>
        <a href="Identity/Account/Manage">
        Hello, @context.User.Identity?.Name!</a>
        <form method="post"
        action="Identity/Account/Logout">
            <button type="submit"
     class="nav-link btn btn-link">
     Log out</button>
```


Log in - Blazor_Server_App
https://localhost:7067/Identity/Account/Login
Blazor_Server_App
Register
Login
Email
Password
☐ Remember me?
Log in
Forgot your password?
Register as a new user
Resend email confirmation

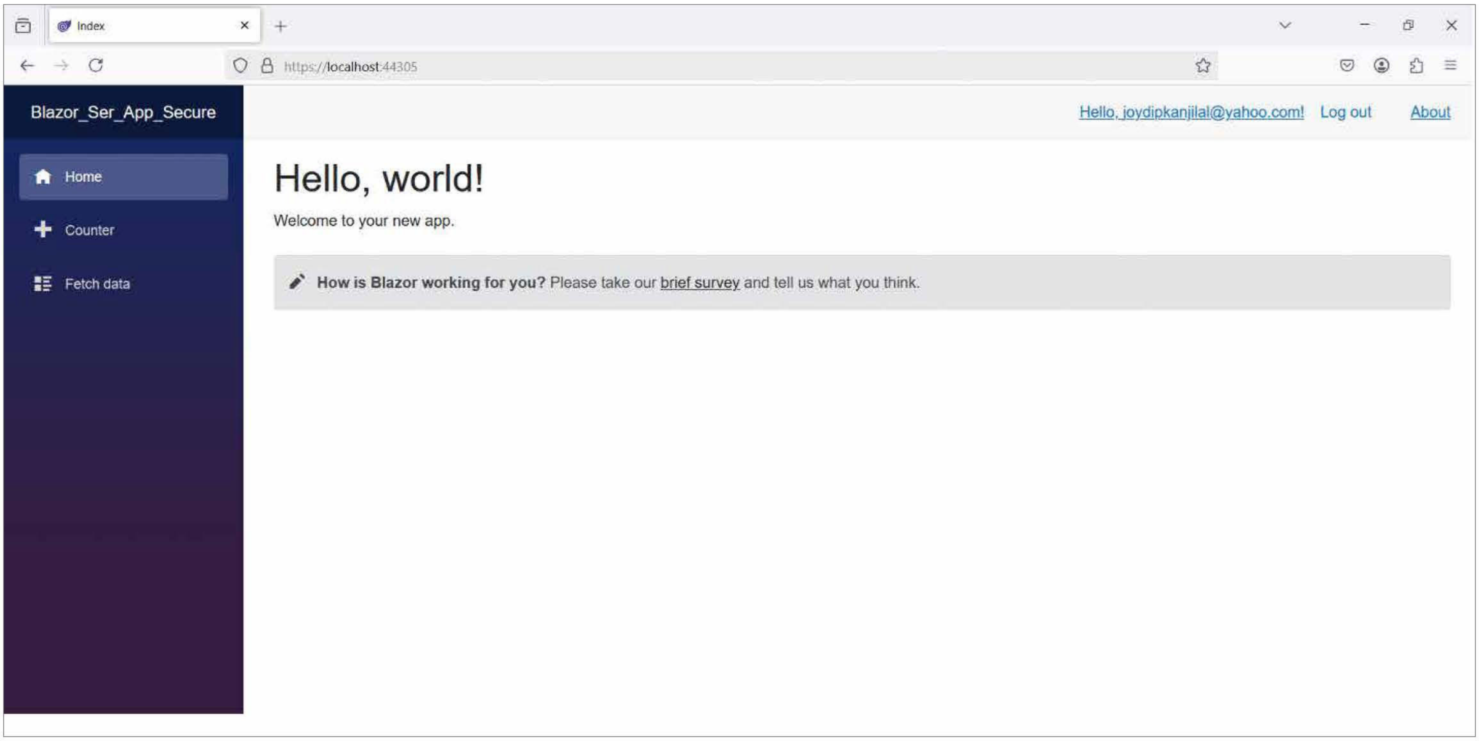
Log in

Use a local account to log in.

Use another service to log in.

There are no external authentication services configured. See this article about setting up this ASP.NET application to support logging in via external services.

Figure 9: The Login Screen


Index
https://localhost:44305
Blazor_Ser_App_Secure
Home
Counter
Fetch data
Hello, joydipkanjilal@yahoo.com!
Log out
About

Hello, world!

Welcome to your new app.

How is Blazor working for you? Please take our brief survey and tell us what you think.

Figure 10: The user is authenticated and logged in to the application.

```
        </form>
    </Authorized>
    <NotAuthorized>
    <a href="Identity/
    Account/Register">Register</a>
        <a href="Identity/
    Account/Login">Log in</a>
    </NotAuthorized>
</AuthorizeView>
```

The AuthorizeView component is comprised of two child components, AuthorizeView and NotAuthorized. The former is used to illustrate that the user is logged in, i.e., the user has been authorized successfully, and the latter is used to show that the user is logged out, i.e., the user is not authorized to access the application.

The Blazor Server application also generates a class named RevalidatingIdentityAuthenticationStateProvider for you. This class extends the RevalidatingServerAuthenticationStateProvider that inherits the ServerAuthenticationStateProvider class, which, in turn, extends the AuthenticationStateProvider abstract class.

Secure a Blazor WebAssembly Application

Blazor WebAssembly (WASM is an acronym for WebAssembly) is a framework for building single-page applications (SPAs) that enables you to execute C# code from client-side applications. It allows you to represent application functionality using C# instead of JavaScript. Blazor WebAssembly applications are comprised of a set of Razor components built using standard C# classes.

In a typical ASP.NET Core hosted model, you'll observe three projects:

- [ProjectName].Server
- [ProjectName].Client
- [ProjectName].Shared

Although the server project hosts the client project, the client project runs directly in the context of a web browser at the client side. When the first request arrives, the compiled application, together with all its dependencies and the .NET Core runtime, are downloaded at the client side.

When you create a standalone Blazor WebAssembly application in .NET 7, you'll observe three projects created automatically for you. These are the Server/API project, the Client project, and a Shared project. You can also create these projects in .NET 9 manually as a workaround. In this example, you'll create the following three projects in the solution:

- BillingSystem.Server
- BilligSystem.Client
- Billing.Shared

The server project is where you usually write the back-end logic (API, services, etc.). The shared project typically contains libraries that are used across both the server and client projects. The client project is where you build your user interface components.

In the next section, I'll examine how you can take advantage of custom authentication strategies to implement security in a real-life Blazor Web Assembly application.

Anatomy of a Blazor WebAssembly Application

A typical Blazor WebAssembly project is comprised of the following folders:

- **wwwroot:** Contains static resources used in your Blazor WebAssembly application

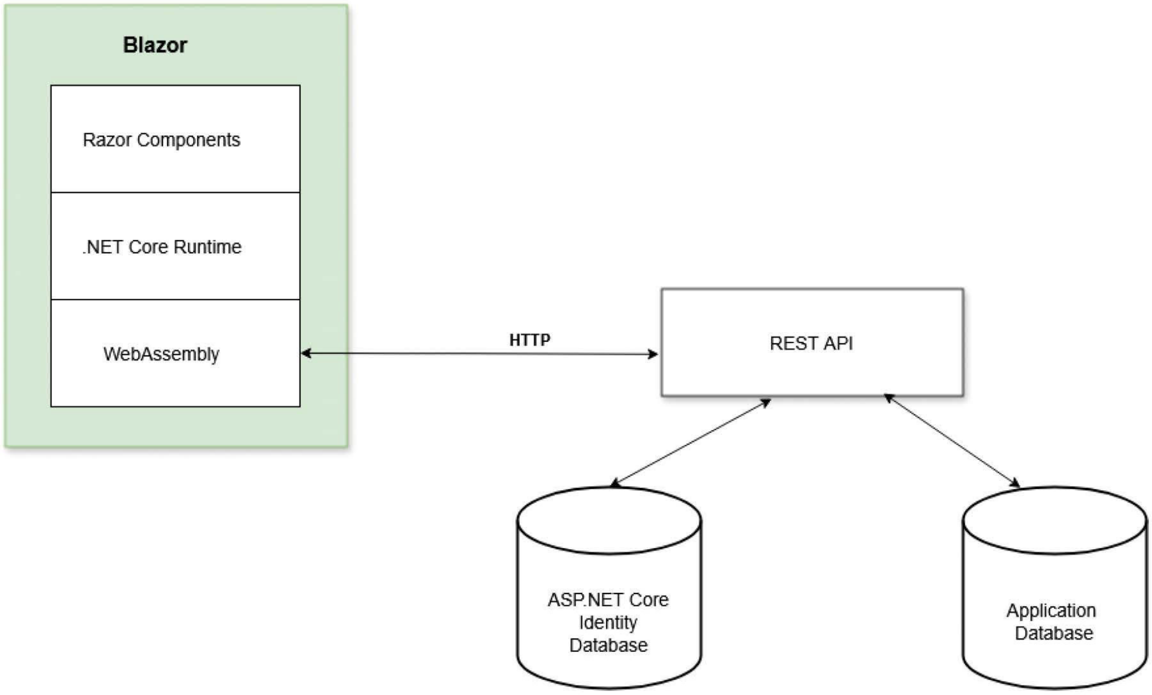


Figure 11: Demonstrating Blazor WebAssembly security at work

- **Shared:** Contains a collection of shared components used in your Blazor WebAssembly application
- **Pages:** The visual elements of your Blazor WebAssembly application

The following is the list of files in a Blazor WebAssembly application:

- **App.razor:** The razor component of your Blazor WebAssembly application
- **_Imports.razor:** A file used to specify the required namespaces used in your Blazor WebAssembly application
- **Program.cs:** The file that contains the start-up code of your Blazor WebAssembly application

The following section examines how to build a microservices-based real-life application using Blazor WebAssembly. This application leverages EF Core to talk to the underlying database for data storage and retrieval.

Implementing a Microservices-Based Billing System

Microservices architecture encompasses a conglomeration of loosely coupled components that can be built using a collection of homogenous or heterogenous technologies. In a typical microservices-based application, there is a single application comprised of a suite of small, independently deployable services, each of which executes and communicates with the others via lightweight communication mechanisms. There's a plethora of benefits of microservices architecture, such as enhanced scalability, improved ROI, faster releases, and reduced coupling.

In this application, you'll build a secure microservices-based Blazor WebAssembly application. To implement security, you can leverage the ASP.NET Core Identity Framework, as shown in **Figure 11**.

Alternatively, you can implement custom authentication and store the security related database tables either in a separate database or in the application database.

Create a New Blazor Web Assembly Project in .NET 9 and Visual Studio 2022

You can create a project in Visual Studio 2022 in several ways, such as, from the Visual Studio 2022 Developer Command Prompt or by launching the Visual Studio 2022 IDE. When you launch Visual Studio 2022, you'll see the Start window. You can choose **Continue without code** to launch the main screen of the Visual Studio 2022 IDE.

Now that you know the basics, let's start setting up the project. To create a new ASP.NET Core 8 Project in Visual Studio 2022:

1. Start the Visual Studio 2022 IDE.
2. In the **Create a new project** window, select **Blazor Web App** and click Next to move on.
3. Specify the project name as **BillingSystem** and the path where it should be created in the **Configure your new project** window.
4. If you want the solution file and project to be created in the same directory, you can optionally check the **Place solution and project in the same directory** checkbox. Click Next to move on.
5. In the next screen, specify the target framework and authentication type as well. Ensure that the **Configure for HTTPS** and **Do not use top-level statements** checkboxes are unchecked because you won't use any of these in this example.
6. Next, specify the **Interactive render mode** and **Interactivity location.**
7. Ensure that the **Include sample pages** checkbox is checked if you would like to have sample pages added to your project
8. Click Create to complete the process.

A new Blazor Web App project is created. **Figure 12** shows the default solution structure.

In this section, you'll build a simple Billing System based off microservices architecture. The types (classes, structs, records, and interfaces) to be used in this application are:

Models
Invoice
Product
Customer
Shipment

Data
ShipmentDbContext
IShipmentRepository
ShipmentRepository

APIs
ShipmentController

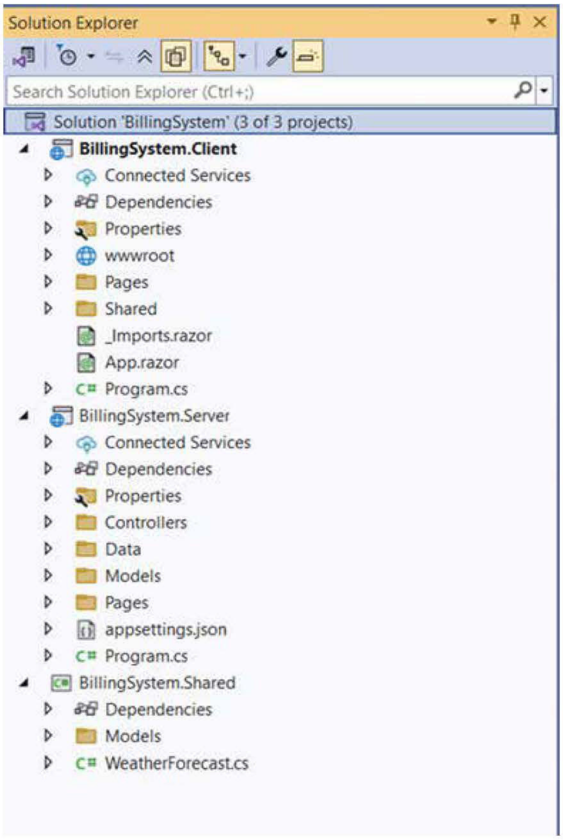

Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'BillingSystem' (3 of 3 projects)
BillingSystem.Client
Connected Services
Dependencies
Properties
wwwroot
Pages
Shared
_Imports.razor
App.razor
Program.cs
BillingSystem.Server
Connected Services
Dependencies
Properties
Controllers
Data
Models
Pages
appsettings.json
Program.cs
BillingSystem.Shared
Dependencies
Models
WeatherForecast.cs

Figure 12: The Solution Explorer of the Blazor WebAssembly application

Listing 1: The database script for our billing system database

```
-- Product table
CREATE TABLE Product (
    Product_Id UniqueIdentifier PRIMARY KEY,
    Product_Name VARCHAR(255) NOT NULL,
    Product_Description TEXT,
    Product_Price DECIMAL(10, 2) NOT NULL
);

-- Customer table
CREATE TABLE Customer (
    Customer_Id UniqueIdentifier PRIMARY KEY,
    Customer_Name VARCHAR(255) NOT NULL,
    Customer_Address VARCHAR(255) NOT NULL,
    Customer_Contact VARCHAR(255),
    Customer_Phone VARCHAR(20),
    Customer_Email VARCHAR(20)
);

-- Invoice table
CREATE TABLE Invoice (
    Invoice_Id UniqueIdentifier PRIMARY KEY,
    Customer_Id UniqueIdentifier NOT NULL,
          Invoice_Amount DECIMAL(10, 2),
    Invoice_Date DATE NOT NULL,
    Payment_Due_Date DATE NOT NULL,
    FOREIGN KEY (Customer_Id) REFERENCES Customer(Customer_Id)
);

-- Invoice_Detail table
CREATE TABLE Invoice_Detail (
    Invoice_Details_Id UniqueIdentifier PRIMARY KEY,
    Invoice_Id UniqueIdentifier NOT NULL,
          Product_Id UniqueIdentifier NOT NULL,
          Discount DECIMAL(10, 2),
    FOREIGN KEY (Invoice_Id) REFERENCES Invoice(Invoice_Id),
          FOREIGN KEY (Product_Id) REFERENCES Product(Product_Id)
);

-- Shipment table
CREATE TABLE Shipment (
    Shipment_Id UniqueIdentifier PRIMARY KEY,
    Invoice_Id UniqueIdentifier NOT NULL,
    Shipment_Address VARCHAR(255) NOT NULL,
    Shipment_Contact VARCHAR(255),
    Shipment_Phone VARCHAR(20),
    Shipment_Date DATE NOT NULL,
    Arrival_Date DATE NOT NULL,
    FOREIGN KEY (Invoice_Id) REFERENCES Invoice(Invoice_Id)
);
```

Create the Logical Model

The following entities make up the logical blueprint for a typical billing system:

- **Invoice:** Can record and maintain all information about each invoice, such as customer ID, invoice date, and payment due date for an invoice.
- **Invoice detail:** Captures and keeps billing information about each transaction for billing purposes.
- **Product:** Captures the goods or services available in any business. The data captured includes product ID, product name, description, price, etc.
- **Customer:** Captures and keeps data on a customer, such as the customer ID, customer name, and other contact details for the customer.

Listing 1 shows the database script for creating the database tables for the billing system. **Figure 13** demonstrates the database diagram of the Billing System database.

The Types Pertaining to the Shipment Microservice

In this example, you'll build the Shipment microservice application or the Shipment API. The shipment microservice application is comprised of the following files:

- **Shipment.cs:** Contains details pertaining to shipment such as invoice ID, contact person, phone number, address, etc.
- **IShipmentRepository.cs:** Represents the IProductRepository interface that contains the declaration of the operations supported by the product repository.

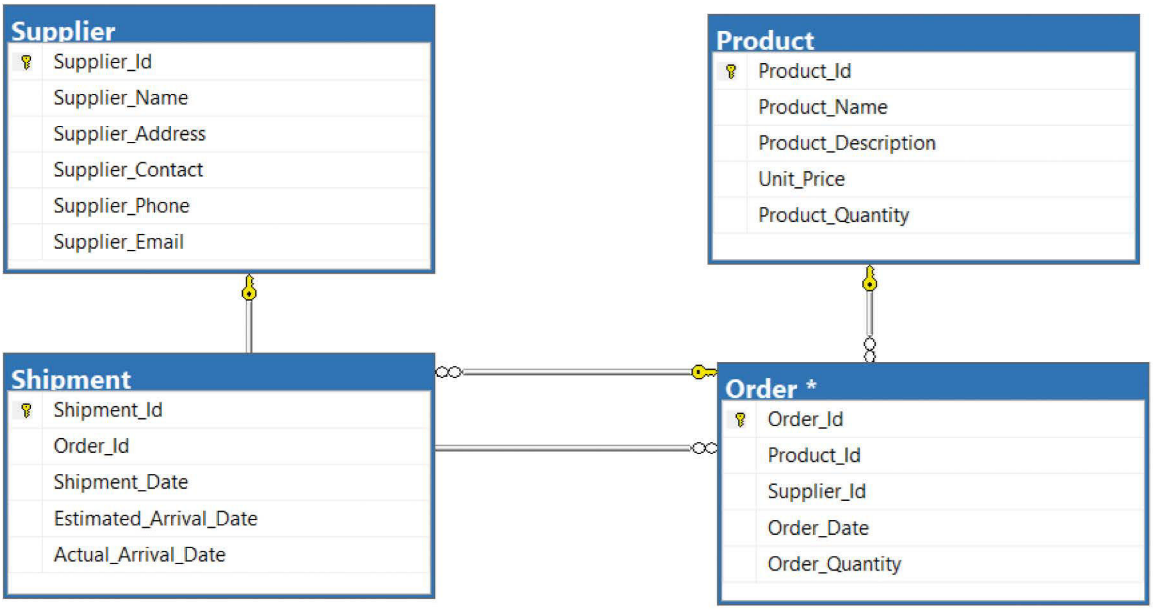

Supplier
Supplier_Id
Supplier_Name
Supplier_Address
Supplier_Contact
Supplier_Phone
Supplier_Email
Product
Product_Id
Product_Name
Product_Description
Unit_Price
Product_Quantity
Shipment
Shipment_Id
Order_Id
Shipment_Date
Estimated_Arrival_Date
Actual_Arrival_Date
Order *
Order_Id
Product_Id
Supplier_Id
Order_Date
Order_Quantity

Figure 13: The database design of the Billing System database

- **ProductRepository.cs:** Implements the members of the IProductRepository interface.
- **ProductDbContext.cs:** The product data context used to perform CRUD operations for the Product table in the database.
- **appsettings.json:** Where you can configure the database connection string, logging metadata, etc.
- **Program.cs:** Any ASP.NET Core application contains a Program.cs file where the startup code required by the application resides and is configured. You can specify dependency injection (DI), configuration, middleware, and other information in this file.

> An application requires a connection string to establish a connection to the database. The connection string contains the database name to which to connect, the instance name of the database server where the database resides, the credentials of the database server, etc.

Specify the Database Connection String

In ASP.NET Core, the application's settings are stored in a file known as appsettings.json. This file is created by default when you create a new ASP.NET Core project. You can take advantage of the ConnectionString property to retrieve or store the connection string for a database. You can specify the connection string in the appsettings.json file, as shown below:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },

  "ConnectionStrings": {
    "SCSDbSettings":
    "Write your connection string here."
  },

  "AllowedHosts": "*"
}
```

You'll use this connection string to enable the application to connect to the database in a section later in this article.

Install Entity Framework Core

So far, so good. The next step is to install the necessary NuGet Package(s) for working with Entity Framework Core and SQL Server. To install these packages into your project, right-click on the solution and then select **Manage NuGet Packages for Solution....**

Once the window pops up, search for the NuGet packages to add to your project. To do this, type in the packages named Microsoft.EntityFrameworkCore, Microsoft.EntityFrameworkCore.Design, Microsoft.EntityFrameworkCore.Tools, and Microsoft.EntityFrameworkCore.SqlServer in the search box of the NuGet Package Manager screen and install them one after the other.

Alternatively, you can type the commands shown below at the NuGet Package Manager Command Prompt:

```
PM> Install-Package
Microsoft.EntityFrameworkCore
PM> Install-Package
Microsoft.EntityFrameworkCore.Design
PM> Install-Package
Microsoft.EntityFrameworkCore.Tools
PM> Install-Package
Microsoft.EntityFrameworkCore.SqlServer
```

You can also install these packages by executing the following commands at the Windows Shell:

```
dotnet add package
Microsoft.EntityFrameworkCore
dotnet add package
 Microsoft.EntityFrameworkCore.Design
dotnet add package
 Microsoft.EntityFrameworkCore.Tools
dotnet add package
 Microsoft.EntityFrameworkCore.
SqlServer
```

Create the Model Classes

First off, create two solution folders named Models and DataAccess. The former will contain one or more model classes and the latter will have the data context and repository interfaces and classes. It should be noted that you can always create multiple data context classes in the same project. If your data context class contains many entity references, it's a good practice to split the data context among multiple data context classes rather than have one large data context class.

> EF Core enables splitting a data context into multiple smaller data contexts, each representing a particular domain or module. However, certain downsides exist, such as the inability to join across contexts and, in some cases, more complicated transaction handling.

Create a new class called Shipment in a file named Shipment.cs inside the Models folder, and write the following code in there:

```
public record Shipment
{
    public Guid Shipment_Id
    { get; set; }

    public Guid Invoice_Id
    { get; set; }

    public string Shipment_Contact
    { get; set; } = default!;

    public string Shipment_Address
    { get; set; } = default!;

    public string Shipment_Phone
    { get; set; } = default!;

    public DateTime Shipment_Date
    { get; set; } = default!;

    public DateTime Arrival_Date
    { get; set; } = default!;
}
```

Create the Data Context

Let's now create the data context class to enable the application to interact with the database to perform CRUD (Create, Read, Update, and Delete) operations. To do this, create a new class named ShipmentDbContext that extends the DbContext class of EF Core, and write the following code in there:

```
public class ShipmentDbContext : DbContext
{
    public DbSet <Models.Shipment>
    Shipments { get; set; }

    protected override void
    OnConfiguring
    (DbContextOptionsBuilder
    optionsBuilder)
    {
        base.OnConfiguring
        (optionsBuilder);
    }
}
```

In the preceding snippet, the statement base.OnConfiguring (optionsBuilder) calls the OnConfiguring method of the base class of your ShipmentDbContext. Because the base class of the ShipmentDbContext class is DbContext, the call does nothing in particular.

> In Entity Framework Core (EF Core), a data context is a component by which an application connects to a database and performs CRUD.

You can specify your database connection string in the OnConfiguring overloaded method of the ShipmentDbContext class. However, in this implementation, you'll store the database connection settings in the AppSettings.json file and read it in the Program.cs file to establish a database connection.

Note that your custom data context class (the ShipmentDbContext class in this example) must expose a public constructor that accepts an instance of type DbContextOptions <ApplicationDbContext> as an argument. This is needed to enable the runtime to pass the context configuration using a call to the AddDbContext() method to your custom DbContext class. The following code snippet illustrates how you can define a public constructor for your data context class:

```
public ShipmentDbContext (DbContextOptions
<ShipmentDbContext> options,
IConfiguration configuration) :  base(options)
{
   _configuration = configuration;
}
```

Seed the Database

You might often want to work with data seeding when using Entity Framework Core (EF Core) to populate a blank database with an initial or minimal data set. Data seeding is a one-time process of loading data into a database. The EF

Listing 2: The ShipmentDbContext class

```
public class ShipmentDbContext : DbContext
{
    private readonly IConfiguration _configuration;
    public ShipmentDbContext(DbContextOptions
    <ShipmentDbContext> options,
      IConfiguration configuration) : base(options)
    {
        _configuration = configuration;
    }

    protected override void OnConfiguring(
    DbContextOptionsBuilder
      optionsBuilder)
    {
        _ = optionsBuilder.UseSqlServer(_configuration.
          GetConnectionString("DefaultConnection")).
          EnableSensitiveDataLogging();
    }

    public DbSet<Shipment> Shipments { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Shipment>().ToTable("Shipment");
        modelBuilder.Entity<Shipment>().HasKey(p =>
          p.Shipment_Id);

        var shipmentData = GenerateShipmentData();
        modelBuilder.Entity<Shipment>().HasData(shipmentData);
    }

    private Shipment[] GenerateShipmentData()
    {
        var invoice_Id = Guid.NewGuid();

        var data = new Faker<Shipment>()
            .RuleFor(x => x.Shipment_Id, f => Guid.NewGuid())
            .RuleFor(x => x.Invoice_Id, invoice_Id)
            .RuleFor(x => x.Shipment_Contact, f => f.Person.FullName)
            .RuleFor(x => x.Shipment_Phone, f => f.Person.Phone)
            .RuleFor(x => x.Shipment_Date, f => DateTime.UtcNow);
        return data.Generate(count: 5).ToArray();
    }
}
```

Core framework provides an easy way to seed the data using the OnModelCreating() method of the DbContext class.

To generate fake data in your ASP.NET Core application, you can take advantage of the Bogus open-source library. It helps to seed your database by taking advantage of randomly generated but realistic data. To use this library, you should install the Bogus library from NuGet into your project.

> In computing, fake data implies data that is artificially generated and looks similar to real-world data. Usually, such data is used for testing software applications.

The following code snippet illustrates how you can generate data using random data using Bogus library:

```
private Shipment[] GenerateShipmentData() {

  var invoice_Id = Guid.NewGuid();
  var data =

      new Faker<Shipment>()
          .RuleFor(x => x.Shipment_Id,
                   f => Guid.NewGuid())
          .RuleFor(x => x.Invoice_Id, invoice_Id)
          .RuleFor(x => x.Shipment_Contact,
                   f => f.Person.FullName)
          .RuleFor(x => x.Shipment_Phone,
                   f => f.Person.Phone)
          .RuleFor(x => x.Shipment_Date,
                   f => DateTime.UtcNow);
  return data.Generate(count: 5).ToArray();
}
```

Invoke the GenerateShipmentData method in the OnModelCreating method to populate the database with randomly generated data as shown in the following piece of code:

```
protected override void OnModelCreating(
    ModelBuilder modelBuilder) {
  modelBuilder.Entity<Shipment>().ToTable(
      "Shipment");
  modelBuilder.Entity<Shipment>().HasKey(
      p => p.Shipment_Id);
  var shipmentData = GenerateShipmentData();
  modelBuilder.Entity<Shipment>().HasData(
      shipmentData);
}
```

The complete source code of the ShipmentDbContext class is given in **Listing 2**.

Create the IShipmentRepository Interface

Create an interface named IShipmentRepository in a file having the name as IShipmentRepository.cs and write the following code in there:

```
public interface IShipmentRepository {
  public Task<List<Shipment>>
  GetShipmentsAsync();
  public Task<Shipment>

  GetShipmentAsync(Guid shipmentId);

  public Task CreateShipmentAsync(
      Shipment shipment);

  public Task UpdateShipmentAsync(
      Shipment shipment);

  public Task DeleteShipmentAsync(
      Guid shipmentId);
}
```

Create the ShipmentRepository Class

Now, create a new class named ShipmentRepository that implements the IShipmentRepository interface in a file having the same name with a .cs extension. Next, write the following code in there:

```
public class ShipmentRepository :
IShipmentRepository
{

}
```

Listing 3 shows how to implement the methods of the IShipmentRepository interface.

The complete source code of the ShipmentRepository class is given in **Listing 4**.

Register the ShipmentRepository Instance

The following code snippet illustrates how an instance of type IShipmentRepository is added as a scoped service to the IServiceCollection in the Program.cs file.

```
builder.Services.AddScoped
<IShipmentRepository, ShipmentRepository>();
```

Create the ShipmentController Class

Now, create a new controller named ShipmentController in the Controllers folder of the project. The following code snippet shows how you can take advantage of constructor injection to pass an instance of type IShipmentRepository and then use it to store and retrieve shipment data to and from the database.

```
[Route("api/[controller]")]
[ApiController]

public class ShipmentController : ControllerBase {
  private IShipmentRepository _shipmentRepository;

  public ShipmentController(
      IShipmentRepository supplierRepository) {
    _shipmentRepository = supplierRepository;
  }
}
```

The following code snippet shows the action methods for creating, updating, and deleting shipment data.

```
[HttpGet("GetShipment")]
public async Task<IActionResult> GetShipment(
```

Listing 3: Implemented methods of the IShipmentRepository interface

```
public async Task<List<Shipment>>
GetShipmentsAsync() {
  return await _shipmentDbContext.Shipments
      .ToListAsync();
}

public async Task<Shipment>
GetShipmentAsync(Guid Id) {
  return await
_shipmentDbContext.Shipments
      .FirstOrDefaultAsync
(x => x.Shipment_Id == Id);
}

public async Task
CreateShipmentAsync(Shipment shipment) {
  await _shipmentDbContext.Shipments.AddAsync(
      shipment);
}

public async Task

UpdateShipmentAsync(Shipment shipment) {
  await _shipmentDbContext.Shipments
      .Where(s => s.Shipment_Id ==
                  shipment.Shipment_Id)

      .ExecuteUpdateAsync(
          s => s.SetProperty(x => x.Shipment_Id,
                             shipment.Shipment_Id)

                   .SetProperty(
                       x => x.Invoice_Id,
                       shipment.Invoice_Id)

                   .SetProperty(
                       x => x.Shipment_Contact,
                       shipment.Shipment_Contact)

                   .SetProperty(
                       x => x.Shipment_Phone,
                       shipment.Shipment_Phone)

                   .SetProperty(
                       x => x.Shipment_Date,
                       shipment.Shipment_Date)

                   .SetProperty(
                       x => x.Arrival_Date,
                       shipment.Arrival_Date)

                   .SetProperty(
                       x => x.Shipment_Address,
                       shipment.Shipment_Address)

                   .SetProperty(
                       x => x.Shipment_Id,
                       shipment.Shipment_Id));
}

public async Task
DeleteShipmentAsync(Guid shipmentId) {
  await _shipmentDbContext.Shipments
      .Where(s => s.Shipment_Id == shipmentId)
      .ExecuteDeleteAsync();
}
```

Listing 4: The ShipmentRepository class

```
public class ShipmentRepository : IShipmentRepository {

  private readonly ShipmentDbContext
      _shipmentDbContext;

  public ShipmentRepository(
      ShipmentDbContext shipmentDbContext) {
    _shipmentDbContext = shipmentDbContext;
    _shipmentDbContext.Database.EnsureCreated();
  }


  public async Task<List<Shipment>> GetShipmentsAsync() {

    return await _shipmentDbContext.Shipments
        .ToListAsync();
  }

  public async Task<Shipment> GetShipmentAsync(Guid Id) {

    return await _shipmentDbContext.Shipments
        .FirstOrDefaultAsync(x => x.Shipment_Id == Id);
  }

  public async Task
  CreateShipmentAsync(Shipment shipment) {
    await _shipmentDbContext.Shipments.AddAsync(
        shipment);
  }

  public async Task
  UpdateShipmentAsync(Shipment shipment) {
    await _shipmentDbContext.Shipments
        .Where(s => s.Shipment_Id ==
                    shipment.Shipment_Id)
        .ExecuteUpdateAsync(
            s =>
                s.SetProperty(
                      x => x.Shipment_Id,
                      shipment.Shipment_Id)
                     .SetProperty(
                         x => x.Invoice_Id,
                         shipment.Invoice_Id)
                     .SetProperty(
                         x => x.Shipment_Contact,
                         shipment.Shipment_Contact)
                     .SetProperty(
                         x => x.Shipment_Phone,
                         shipment.Shipment_Phone)
                     .SetProperty(

                         x => x.Shipment_Date,
                         shipment.Shipment_Date)

                     .SetProperty(
                         x => x.Arrival_Date,
                         shipment.Arrival_Date)

                     .SetProperty(

                         x => x.Shipment_Address,
                         shipment.Shipment_Address)

                     .SetProperty(

                         x => x.Shipment_Id,
                         shipment.Shipment_Id));
  }

  public async Task DeleteShipmentAsync(Guid shipmentId) {
    await _shipmentDbContext.Shipments
        .Where(s => s.Shipment_Id == shipmentId)
        .ExecuteDeleteAsync();
  }
}
```

Copilot • AI • ChatGPT • .NET • ASP.NET • GitHub • Visual Studios • Open AI • LLM • Azure AI Studio • and more…

DEVintersection
CONFERENCE

Cybersecurity INTERSECTION & next GenAI CONFERENCE

OCT 7-9, 2025 *WORKSHOPS OCT 5, 6 & 10*
LOEWS SAPPHIRE FALLS RESORT *UNIVERSAL ORLANDO*

Learn from Microsoft Leaders, Engineers and Industry Experts:

SCOTT HANSELMAN
Vice President, Developer Community, Microsoft

SCOTT HUNTER
Vice President, Director Product Management for Azure Developer Experience, Microsoft

MADDY MONTAQUILA
Senior Program Manager, .NET MAUI, Microsoft

SAFIA ABDELLA
Senior Software Engineer - .NET, Microsoft

JEFF FRITZ
Principal Program Manager, Microsoft

JOHN PAPA
Partner GM - Cloud Advocacy, Microsoft

MARKUS EGGER
President and Chief Software Architect, CODE Group

ZOINER TEJADA
CEO and Architect, Solliance

MICHELE LEROUX BUSTAMANTE
President & Architect, Solliance

RICHARD CAMPBELL
Advisor, Creator, Storyteller, Campbell & Associates

REGISTER TODAY!

DEVintersection.com

BONUS: CODE Magazine Track

Questions Answered, Strategies Defined, Relationships Built

```
    Guid shipmentId) {

  return Ok(
      await _shipmentRepository.GetShipmentAsync(
          shipmentId));
}

[HttpGet("GetShipments")]
public async Task<IActionResult> GetShipments() {
  return Ok(await _shipmentRepository
                .GetShipmentsAsync());
}

[HttpPost("CreateShipment")]
public async Task<IActionResult> CreateOrder(
    [FromBody] Shipment shipment) {
  await _shipmentRepository.CreateShipmentAsync(
      shipment);
  return Ok(shipment.Shipment_Id);
}

[HttpPost("UpdateShipment")]
public async Task<IActionResult> UpdateOrder(
    [FromBody] Shipment shipment) {
  await _shipmentRepository.UpdateShipmentAsync(
      shipment);
  return NoContent();
}

[HttpDelete("DeleteShipment{id}")]
public async Task<IActionResult> DeleteShipment(
    Guid shipmentId) {
  await _shipmentRepository.DeleteShipmentAsync(
      shipmentId);
  return NoContent();
}
```

The complete source code of the ShipmentController class is given in **Listing 5**.

Execute the Billing System Application

Finally, press F5 while you're inside the Visual Studio 2022 IDE to run the application. **Figure 14** shows how the home page of the Billing System application looks in your web browser.

In the next section, I'll examine how you can secure the Billing System application so that only authorized users can access it.

Secure the Billing System Application

In this section, I'll examine how you can secure the Billing System application to thwart any unauthorized requests to access the application. You'll use the following types (classes, structs, records, and interfaces) to implement the security module of the BillingSystem application.

Models
ApplicationUser

Data
ApplicationDbContext
IShipmentRepository
ShipmentRepository

APIs
AuthenticationController

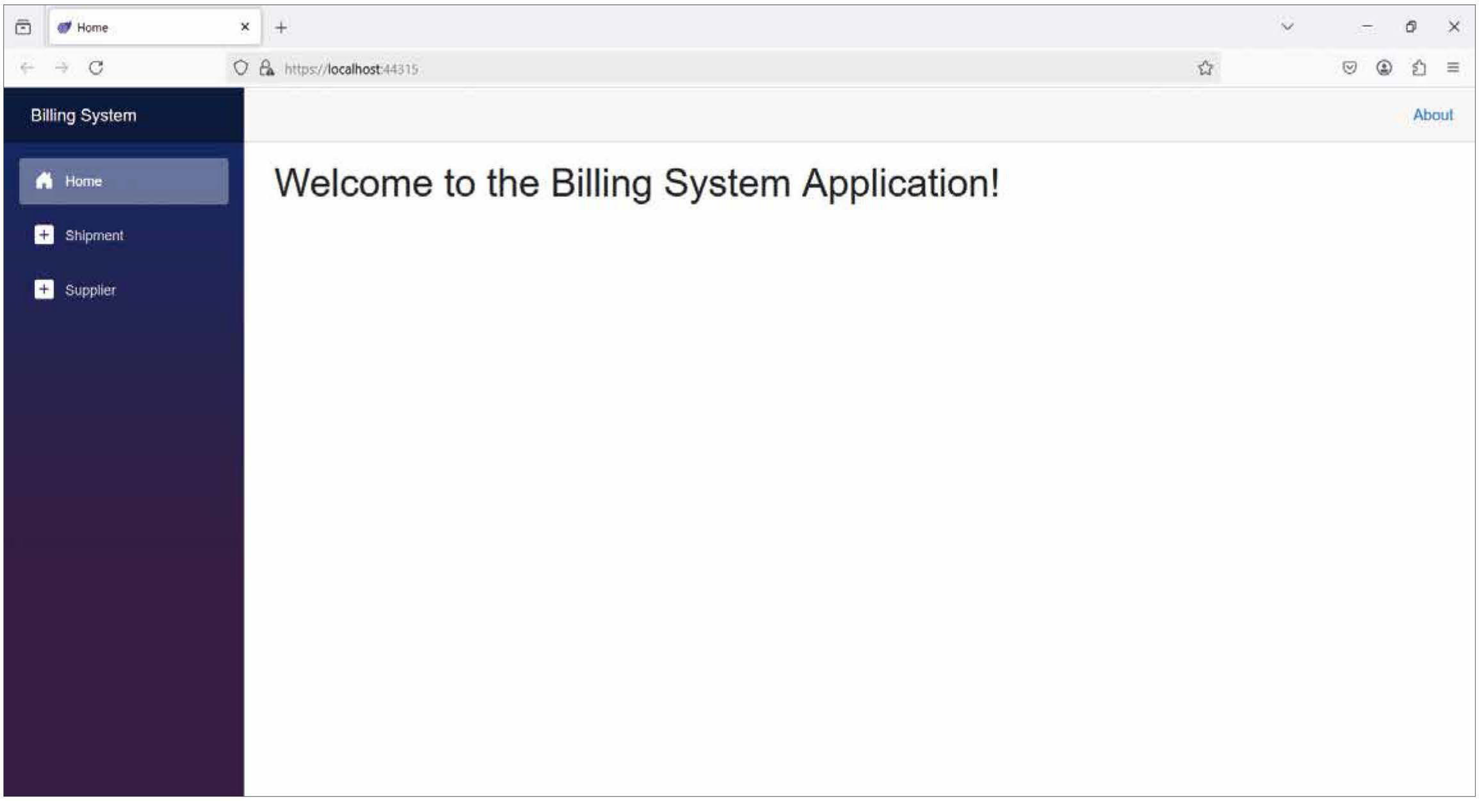
Create the Models

Create a new solution folder named Models inside the BillingSystem project you created earlier. Create a model class named ApplicationUser that extends the IdentityUser class of the ASP.NET Core Identity framework and write the following piece of code in there:

```
using Microsoft.AspNet.Identity.EntityFramework;

namespace BillingSystem.Models
{
```

Listing 5: The ShipmentController class

```
[Route("api/[controller]")]
[ApiController]

public class ShipmentController : ControllerBase {
  private IShipmentRepository _shipmentRepository;
  public ShipmentController(
      IShipmentRepository supplierRepository) {
    _shipmentRepository = supplierRepository;
  }

  [HttpGet("GetShipment")]
  public async Task<IActionResult> GetShipment(
      Guid shipmentId) {
    return Ok(await _shipmentRepository
                  .GetShipmentAsync(shipmentId));
  }

  [HttpGet("GetShipments")]
  public async Task<IActionResult>
  GetShipments() {
    return Ok(await _shipmentRepository
                  .GetShipmentsAsync());

  }

  [HttpPost("CreateShipment")]
  public async Task<IActionResult> CreateOrder(
      [FromBody] Shipment shipment) {
    await _shipmentRepository.CreateShipmentAsync(
        shipment);

    return Ok(shipment.Shipment_Id);
  }

  [HttpPost("UpdateShipment")]
  public async Task<IActionResult> UpdateOrder(
      [FromBody] Shipment shipment) {
    await _shipmentRepository.UpdateShipmentAsync(
        shipment);

    return NoContent();
  }

  [HttpDelete("DeleteShipment{id}")]
  public async Task<IActionResult> DeleteShipment(
      Guid shipmentId) {
    await _shipmentRepository.DeleteShipmentAsync(
        shipmentId);
    return NoContent();
  }
}
```



Welcome to the Billing System Application!

Figure 14: The Billing System application at work

```
    public class ApplicationUser : IdentityUser
    {

    }
}
```

Next, create a class library project called BillingSystem.Shared. Inside this project, create a solution folder named Models and create the LoginRequest.cs with the following code in it:

```
using System.ComponentModel.DataAnnotations;

namespace BillingSystem.Shared.Models
{
    public class LoginRequest
    {
        [Required]
        public string UserName { get; set; }

        [Required]
```

Listing 6: The Login.razor file

```
@page "/login"
@using BillingSystem.Shared.Models
<h3>Login</h3>

<EditForm Model="loginRequest"
OnValidSubmit="OnLoginUser">
   <DataAnnotationsValidator />

   <div class="form-group">
       <label for="username">Username:</label>
       <InputText id="username" class="form-control"
       @bind-Value="@loginRequest.Username" />
   </div>

   <div class="form-group">
       <label for="password">Password:</label>
       <InputText id="password" type="password"
class="form-control"
      @bind-Value="@loginRequest.Password" />
   </div>

   <button type="submit"
class="btn btn-primary">Login</button>
   <label class="text-danger">@error</label>
   <NavLink href="register">
       <h6 class="font-weight-normal text-center">
       Create new user</h6>
   </NavLink>
</EditForm>

@code {
   LoginRequest loginRequest { get; set; } =
new LoginRequest();

   async Task OnLoginUser()
   {
       try
       {
           //Code to authenticate the user
       }

       catch (Exception ex)
       {
           //Code to handle exceptions
       }
   }
}
```

Listing 7: The Register.razor file

```
@page "/register"
@using BillingSystem.Shared.Models
<h3>Register</h3>

<EditForm Model="@registerModel"
OnValidSubmit="@OnRegisterUser">
    <DataAnnotationsValidator />
    <ValidationSummary />

    <div class="form-group">
        <label for="username">Username:</label>
        <InputText id="username" class="form-control"
       @bind-Value="@registerRequest.Username" />
    </div>

    <div class="form-group">
        <label for="password">Password:</label>
        <InputText id="password"
type="password" class="form-control"
       @bind-Value="@registerRequest.Password" />
    </div>

    <div class="form-group">
        <label for="confirmpassword">
Confirm Password:</label>
        <InputText id="confirmpassword"
type="password"
       class="form-control"
@bind-Value="registerRequest.ConfirmPassword" />
    </div>

    <button type="submit"
class="btn btn-success">Register</button>
</EditForm>

@code {
   RegisterRequest registerRequest
   { get; set; } = new   RegisterRequest();

   async Task OnRegisterUser()
    {
        try
        {
            //Code to register a new user
        }

        catch (Exception ex)
        {
            //Code to handle exceptions
        }
    }
}
```

```
        public string Password { get; set; }
    }
}
```

Create a new class named RegisterRequest in a file named RegisterRequest.cs and write the following code in there:

```
using System.ComponentModel.DataAnnotations;

namespace BillingSystem.Shared.Models
{
    public class RegisterRequest
    {
        [Required]
        public string UserName { get; set; }

        [Required]
        public string Password { get; set; }

        [Required]
        [Compare(nameof(Password),
        ErrorMessage =
        "Passwords do not match!")]
        public string ConfirmPassword
        { get; set; }
    }
}
```

Create the User Interface

Create a new file called Login.razor that generates the user interface for enabling a user to log in to the application. **Listing 6** shows the complete source code of the Login.razir file.

Create another new file named Register.razor to register a new user. **Listing 7** shows the complete source code of the Register.razor file.

An Introduction to the ASP.NET Core Identity Framework

Identity Server is an open-source framework for implementing identity and access control in.NET applications. It implements OpenID Connect (OIDC) and OAuth 2.0 standards and integrates with the ASP.NET Core Identity framework and provides a common way to authenticate requests in ASP.NET Core applications. ASP.NET Core Identity is a built-in membership system used to implement log-in functionality in an application. It provides several features such as user authentication, roles- and claims-based authorization, integration with external providers such as Facebook, Google, etc.

> When building web apps that use the ASP.NET Core Identity Framework, user management interfaces in the application should use Razor Pages instead of Razor components. Although it's possible, Microsoft does not endorse or provide support for Razor components handling identity operations.

You can use the ASP.NET Core Identity Framework for creating, updating, deleting, and querying user accounts. You can use this framework to implement register, log in, and log out functionality in your ASP.NET Core application seamlessly. Blazor leverages ASP.NET Core's authentication mechanism to establish a user's identity. However, contrary to the Blazor communication model, the ASP.NET Core Identity Framework is designed to use the HTTP request-response pattern.

Create the Application Data Context

You'll now create a data context named ApplicationDbContext inside a file named ApplicationDbContext.cs and write the following code in there:

```
public class ApplicationDbContext :
IdentityDbContext<ApplicationUser>
{

   public ApplicationDBContext(DbContextOptions
   <ApplicationDbContext> options) : base(options)
   {

   }
}
```

You should now define the connection string to be used to connect to the database and add EF Core capabilities to your application. To do this, write the following code in the Program.cs file:

```
services.AddDbContext<ApplicationDBContext>
(options => options.UseSqlServer
(Configuration.GetConnectionString
("DefaultConnection")));
```

Configure the Identity Framework for Your Blazor WebAssembly Application

Add the following statements in the Program.cs file of the server project to set up identity framework.

```
services.AddIdentity
<ApplicationUser, IdentityRole>().AddEntityFrameworkStores
<ApplicationDBContext>();
builder.Services.AddDefaultIdentity<IdentityUser>
(options => options.SignIn.
RequireConfirmedAccount = true)
    .AddEntityFrameworkStores
<ApplicationDbContext>();
```

Using a Single Data Context

While working with EF Core, you can create one or more data contexts for your application. In this example, I've created two data contexts:

- ApplicationDbContext
- ShipmentDbContext

You can also merge the source code of all your data context classes into the ApplicationDbContext class so that you have only one data context in the application. To do this, overwrite the source code of the ApplicationDbContext class and merge the source code of the ShipmentDbContext class inside it. The complete source code of the merged ApplicationDbContext class is given in **Listing 8**.

Listing 8: The merged ApplicationDbContext class

```
using BillingSystem.Server.Models;
using BillingSystem.Shared.Models;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore;
using Bogus;

namespace BillingSystem.Server.Data
{
    public class ApplicationDbContext :
    IdentityDbContext<ApplicationUser>
    {
        private readonly IConfiguration _configuration;
        public ApplicationDbContext
        (DbContextOptions<ApplicationDbContext> options) :
        base(options)
        {

        }

        public DbSet<Shipment> Shipments { get; set; }

        protected override void OnConfiguring
            (DbContextOptionsBuilder optionsBuilder)
        {
            _ = optionsBuilder.UseSqlServer(_configuration.
              GetConnectionString("DefaultConnection")).
              EnableSensitiveDataLogging();
        }

        protected override void OnModelCreating(
        ModelBuilder modelBuilder)
        {
            base.OnModelCreating(modelBuilder);
            modelBuilder.Ignore<Microsoft.AspNetCore.Identity.
            IdentityUserLogin<string>>();
            modelBuilder.Ignore<Microsoft.AspNetCore.Identity.
            IdentityUserRole<string>>();
            modelBuilder.Ignore<Microsoft.AspNetCore.Identity.
            IdentityUserClaim<string>>();

            modelBuilder.Ignore<IdentityUserToken<string>>();
            modelBuilder.Ignore<IdentityUser<string>>();
            modelBuilder.Ignore<ApplicationUser>();

            modelBuilder.Entity<Shipment>().ToTable("Shipment");
            modelBuilder.Entity<Shipment>().HasKey(p => p.Shipment_Id);
            var shipmentData = GenerateShipmentData();
            modelBuilder.Entity<Shipment>().HasData(shipmentData);
        }

        private Shipment[] GenerateShipmentData()
        {
            var invoice_Id = Guid.NewGuid();
            var data = new Faker<Shipment>()
                .RuleFor(x => x.Shipment_Id, f => Guid.NewGuid())
                .RuleFor(x => x.Invoice_Id, invoice_Id)
                .RuleFor(x => x.Shipment_Contact, f => f.Person.FullName)
                .RuleFor(x => x.Shipment_Phone, f => f.Person.Phone)
                .RuleFor(x => x.Shipment_Date, f => DateTime.UtcNow);

            return data.Generate(count: 5).ToArray();
        }
    }
}
```

Build the Custom Authentication Provider

You can build your custom authentication state provider class by extending the AuthenticationStateProvider or any of the other classes in the inheritance hierarchy. The following code snippet illustrates how you can build your custom authentication state provider class by extending the AuthenticationStateProvider class:

```
using Microsoft.AspNetCore.
Components.Authorization;
using System.Security.Claims;

namespace Blazor_Server_App.Areas.Identity
{
    public class
    CustomAuthenticationStateProvider :
    AuthenticationStateProvider
    {
        public override Task<AuthenticationState>
        GetAuthenticationStateAsync()
        {

             throw new NotImplementedException();

        }
    }
}
```

The following code snippet illustrates how you can modify the GetAuthenticationAsync method to retrieve the current user:

```
public async override Task<AuthenticationState>
GetAuthenticationStateAsync()
{
   try
   {
        var identity = new ClaimsIdentity(new[]
        {
             new Claim(ClaimTypes.Name, "Joydip"),
             new Claim(ClaimTypes.Role, "Admin")

        }, null);

         return await Task.FromResult
         (new AuthenticationState
         (new ClaimsPrincipal(identity)));
   }

   catch(Exception ex)
   {
       Console.WriteLine
       ("Error retrieving authentication state:" +
       ex.ToString());
   }

    return null;
}
```

The following code snippet shows how you can implement a method named Logout. This method will be used to log out the current logged-in user in your application:

```
public async Task<AuthenticationState> Logout()
{
       this.CurrentUser = null;
       return await GetAuthenticationStateAsync();
}
```

Register your CustomAuthenticationStateProvider class in the Program.cs file using the following piece of code:

```
builder.Services.AddScoped
<AuthenticationStateProvider, CustomAuthenticationStateProv
ider>();
```

The complete source code of the CustomAuthenticationProvider class is given in **Listing 9**.

Listing 9: The CustomAuthenticationStateProvider class

```
public class CustomAuthenticationStateProvider :
AuthenticationStateProvider
    {
        private ClaimsPrincipal CurrentUser { get; set; }
        public async override Task<AuthenticationState>
        GetAuthenticationStateAsync()
        {
            try
            {
                var identity = new ClaimsIdentity(new[]
                {
                    new Claim(ClaimTypes.Name, "Joydip"),
                    new Claim(ClaimTypes.Role, "Admin")
                }, null);

                return await Task.FromResult(
                new AuthenticationState
                (new ClaimsPrincipal(identity)));
            }

            catch(Exception ex)
            {
                Console.WriteLine
                ("Error retrieving
                authentication state:" +
                ex.ToString());
            }

            return null;
        }
    }
```

Create the AuthenticationController Class

You need an API endpoint to log in and log out an existing user and also register a new user. To achieve this, you'll create an API controller in the server project named AuthentcationController. The actions methods of this controller class will take advantage of the ASP.NET Core Identity Framework.

The following code snippet illustrates the Login action method that can be used to lo gin an existing user to the application.

```
[HttpPost]
public async Task<IActionResult> Login
(LoginRequest request)
{
    var user = await _userManager.
    FindByNameAsync(request.UserName);

    if (user == null)
        return BadRequest
    ("The user does not exist");
```

Listing 10: The AuthenticationController class

```
using BillingSystem.Server.Models;
using BillingSystem.Shared.Models;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;

namespace BillingSystem.Server.Controllers
{
    [Route("api/[controller]")]
    [ApiController]

    public class AuthenticationController : ControllerBase
    {

        private readonly UserManager
<ApplicationUser> _userManager;
        private readonly SignInManager<ApplicationUser> _signInManager;
        public AuthenticationController
        (UserManager<ApplicationUser> userManager,
            SignInManager<ApplicationUser> signInManager)
        {
            _userManager = userManager;
            _signInManager = signInManager;
        }

        [HttpPost]
        public async Task<IActionResult>
Login(LoginRequest request)
        {
            var user = await _userManager.FindByNameAsync(request.UserName);

            if (user == null)
                return BadRequest("The user does not exist");

            var result = await _signInManager.CheckPasswordSignInAsync
                (user, request.Password, false);

            if (!result.Succeeded)
                return BadRequest("Invalid password");
            await _signInManager.SignInAsync(user, true);

            return Ok(request.UserName);
        }

        [HttpPost]
        public async Task<IActionResult> Register
        (RegisterRequest registerRequest)
        {

            var user = new ApplicationUser();
            user.UserName = registerRequest.UserName;
            var result = await _userManager.CreateAsync
            (user, registerRequest.Password);

            if (!result.Succeeded)
                return BadRequest("Error registering user");

            return Ok(registerRequest.UserName);
        }

        [Authorize]
        [HttpPost]
        public async Task<IActionResult> Logout()
        {
            await _signInManager.SignOutAsync();
            return Ok();
        }
    }
}
```

```
    var result = await _signInManager.
     CheckPasswordSignInAsync
        (user, request.Password, false);



    if (!result.Succeeded)
        return BadRequest("Invalid password");
    await _signInManager.SignInAsync(user, true);

    return Ok(request.UserName);
}
```

The Register action method given below is used to register a new user in the application.

```
[HttpPost]
public async Task<IActionResult> Register
(RegisterRequest registerRequest)
{

    var user = new ApplicationUser();
    user.UserName = registerRequest.UserName;
    var result = await _userManager.CreateAsync
    (user, registerRequest.Password);

   if (!result.Succeeded)
        return BadRequest
   ("Error registering user");
   return Ok(registerRequest.UserName);
 }
```

The Logout action method will be used to log out an already logged-in user in the application.

```
[Authorize]
[HttpPost]
public async Task<IActionResult> Logout()
{
    await _signInManager.SignOutAsync();
    return Ok();
}
```

The complete source code of the AuthenticationController class is given in **Listing 10**.

Add Authentication and Authorization Middleware to Your Application

Ensure that you add the following two statements in the Program.cs file to add the Authentication and Authorization middleware to the request processing pipeline.

```
app.UseAuthentication();
app.UseAuthorization();
```

The complete source code of the Program.cs file is given in **Listing 11**.

Secure Blazor Applications Using Anti-forgery Tokens

By using anti-forgery tokens, you can prevent CRSF attacks on your ASP.NET Core Blazor applications. If you're using Blazor WebAssembly with ASP.NET Core Identity

Listing 11: The Program.cs file

```
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using BillingSystem.Server.Data;
using BillingSystem.Server.Models;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddControllers();
builder.Services.AddDbContext<ApplicationDbContext>
(options => options.UseSqlServer
(Configuration.GetConnectionString("DefaultConnection")));

builder.Services.AddIdentity<ApplicationUser,
IdentityRole>().AddEntityFrameworkStores<ApplicationDbContext>();
builder.Services.AddDefaultIdentity<IdentityUser>(options =>
{
    options.SignIn.RequireConfirmedAccount = false;
    options.Lockout.MaxFailedAccessAttempts = 10;
    options.Lockout.DefaultLockoutTimeSpan = TimeSpan.FromMinutes(5);
}).AddEntityFrameworkStores<ApplicationDbContext>();

builder.Services.AddScoped<IShipmentRepository,
ShipmentRepository>();

var app = builder.Build();

// Configure the HTTP request pipeline.

app.UseHttpsRedirection();
app.UseAuthorization();
app.MapControllers();

app.Run();
```

framework, you should implement anti-forgery tokens in your HttpPost action methods, as shown in the code snippet given below:

```
[HttpGet("getantiforgerytoken")]
public IActionResult GetAntiforgeryToken()
{
    var tokens = _antiforgery.
    GetAndStoreTokens(HttpContext);
    return new ObjectResult(new {
    requestToken = tokens.RequestToken });
}
```

Then retrieve this token at the client side of your Blazor WebAssembly application, store it, and pass this token in all subsequent HTTP POST requests. If you're using a Blazor Server application, pass an anti-forgery token each time a form is submitted.

Best Practices

Safeguarding your Blazor applications is imperative in averting data breaches and ensuring data security. To secure your Blazor applications efficiently, you need consistent updates and strategies for addressing specific security concerns. Follow the recommended practices to enable your developers to design reliable and secure Blazor applications that are resilient to known vulnerabilities and threats.

Here are the key strategies that can be implemented to secure your Blazor efficiently:

- **Storing API Keys:** API keys should never be included in the application code. Instead, they should be stored in environment variables or secure configuration files. It’s a recommended practice to use user secrets while developing your application and using Azure Key Vault to store them when your application is live in production.
- **Update and patch regularly:** Always keep Blazor and other dependencies updated to the newest version, ensuring their security optimizations and changes are correctly applied. Also monitor any third-party libraries or frameworks that your application uses and update them at regular intervals.
- **Protect your APIs:** To safeguard your application's APIs, consider using HTTPS and enforcing strict data validation for inputs. Also implement CORS policies to restrict access to your APIs and use an appropriate authentication mechanism, such as token-based authentication, to secure your API endpoints. You can also implement rate-limiting to prevent API abuse, update the Blazor assemblies regularly, and implement appropriate logging and monitoring strategies to detect any security issues.
- **Using authentication and authorization:** Authentication aims to verify that a person is who they claim to be, i.e., that the credentials they use for logging into the system are genuine and can be trusted. You can choose strategies such as JWT tokens, OAuth tokens, or API keys to implementing authentication. Authorization refers to the process by which the application determines what resources are accessible to an authenticated user and what information they can access within the application. You can use several techniques to implement authorization, such as attribute-based or role-based access control systems.
- **Monitor and log:** Take advantage of a comprehensive approach to handling exceptions in your application. Your logs should capture key details required for diagnosing problems in your code while avoiding sensitive information. Also review the logs for suspicious activities that may present a security problem. To log data, you may use the built-in logging capabilities provided by ASP.NET Core or you can integrate a more advanced and versatile logging framework, such as Serilog.
- **Input validation:** Another way to protect APIs is by verifying that the input data, i.e., the data entered by the user is valid and satisfies specific requirements. To secure your API, you must validate the input data, i.e., the user's entered data. This helps you protect your API from injection attacks that include cross-site scripting (XSS) and SQL Injection.
- **Data protection**: Take advantage of the ASP.NET Core Data Protection API to protect sensitive information in your application. Also use strong hashing algorithms to hash passwords. Never store sensitive information in local or session browser storage. If you have to use these storage options, always encrypt the data.

> Rate limiting restricts the number of requests a client may make to a sever in a particular time frame. It can help thwart denial-of-service attacks and brute-force attacks by limiting the back requests made to your APIs in a certain time period.

- **Use HTTPS:** Always use HTTPS to protect your API endpoints. Ensure that you've used appropriate access controls to safeguard sensitive data exposed by your Blazor applications. Use the HTTPS protocol to secure data in transit and ensure that API keys are updated periodically.
- **Using timeouts:** For several reasons, when you're working with applications that leverage APIs, the downstream services might need to be fixed, i.e., they're non-responsive. Possible causes can be because the services are processing data, it's taking quite a lot of time to read from the database, etc. Whatever the reason, don't allow your API to hang for too long while the clients wait for a response. To prevent this, you can leverage timeouts for your slow or non-responsive services.
- **Secure deployment:** Ensure that you have configured the server and application settings correctly and removed all debugging and error messages in the production environments. Select an appropriate deployment platform for deploying your Blazor applications such as AWS, Azure App Service, and Google Cloud, that not only supports Blazor applications but also provides security features.

> Cross-site scripting (XSS) and Cross-Site Request Forgery (CSRF) are typical examples of web security vulnerabilities. They differ in how they operate. The former enables the attacker to place and run malicious scripts within the context of victim's web browser. In the latter, the attacker deceives the victim user into executing commands the user did not mean to execute.

Where Do You Go from Here?

To guard your application against security threats and vulnerabilities and to protect your data from misuse, you can implement several strategies, such as authentication, authorization, API security, input validation, and shielding your application against XSS and CSRF attacks.

The back-end-for-front-end pattern (for more about this, see my previous article: https://www.codemag.com/Article/2203081/A-Deep-Dive-into-the-Back-End-for-Front-End-Pattern) provides several security benefits to your Blazor application. It enables your server component to manage security functions, like token generation and management while blocking direct API communication.

Implementing application security should not be a one-time exercise. Instead, it should be an ongoing process that requires you to learn and adapt new tools, technologies, techniques, and security guidelines. Finding security flaws or vulnerabilities within applications that attackers can exploit isn't as simple as it seems. You can adhere to the recommended guidelines and devise a strategy to implement strong security and shield your application from prying eyes. With the right mindset and appropriate security strategies, you can build secure and scalable Blazor applications.

Joydip Kanjilal

SPONSORED SIDEBAR:

CODE Is Hiring!

CODE Staffing is accepting resumes for various open positions ranging from junior to senior roles.

We have **multiple openings** and will consider candidates who seek full-time employment or contracting opportunities. **For more information:** www.codestaffing.com.

ONLINE QUICK ID 2560561

Unlock Serverless Azure Functions

The evolution of cloud computing has transformed how developers build and deploy applications. One of the most significant shifts in this evolution is the rise of serverless architecture. Traditionally, building and maintaining applications requires managing servers, scaling infrastructure, and handling operational concerns. With serverless architecture, the cloud provider handles these tasks, allowing developers to focus entirely on writing and deploying code.

Vaibhav Gujral
vaibhav@vaibhavgujral.com

Vaibhav Gujral is a Cloud Architect with over two decades of IT experience. He excels in architecting scalable, cutting-edge cloud ecosystems, blending DevOps principles and AI innovation to drive transformative solutions.

A five-time Microsoft MVP in Azure and AI Platform, Vaibhav is a recognized leader in his field, a Forbes Technology Council member and Senior Member of IEEE.

Through his blog (https://blog.vaibhavgujral.com) and speaking engagements, he shares deep insights with a global audience. Outside his professional pursuits, he's intrigued by how technology can solve real-world challenges.

Azure Functions serves as Microsoft's serverless computing service, enabling developers to execute code in response to various triggers, such as HTTP requests, timers, handling Azure Storage events, working with messaging services like Service Bus or events from other Azure services, and enabling you to build versatile, scalable applications. With Azure Functions, you only pay for the time your code runs, making it highly cost-efficient for event-driven and burst workloads. Its flexibility and simplicity make Azure Functions ideal for microservices, API development, background tasks, and more scenarios.

In this article, I'll explore the fundamentals of Azure Functions. You'll learn how to set up a basic function using Visual Studio Code, understand the critical concepts of triggers and bindings, and create a simple HTTP-triggered function. You'll also learn about handling errors, optimizing performance, integrating Azure Functions with other Azure services, and advanced use cases for Azure Functions. I'll also discuss security best practices and future trends to help you build secure, scalable, and efficient serverless applications.

Whether you're new to cloud development or an experienced developer looking to streamline your workloads, Azure Functions offers a powerful and flexible solution for modern cloud-native applications.

Serverless Architecture and Azure Functions

Serverless architecture shifts infrastructure management from the developer to the cloud provider. Although servers still exist, developers no longer need to manage them directly. Instead, they focus on writing and deploying code while the cloud provider handles server provisioning, scaling, and maintenance.

Serverless computing offers several key advantages:

- **No infrastructure management**: You focus solely on writing code without worrying about servers or infrastructure.
- **Automatic scaling**: Applications automatically scale based on incoming traffic and workload demands.
- **Cost-efficiency**: You only pay for the actual execution time of your functions rather than maintaining always-on servers.
- **Focus on code**: Developers can concentrate on business logic while the platform manages infrastructure tasks like scaling and patching.

Common uses for Azure Functions include:

- **Microservices**: Break down complex applications into smaller, independent services.
- **APIs**: Quickly deploy and scale APIs without managing servers.
- **Event-driven workflows**: Respond to events such as file uploads, database updates, or queue messages.
- **Background processing**: Run long-running tasks like image processing or sending notifications.

Setting Up a Basic Azure Function

Let's move on to setting up your first Azure Function. You'll use Visual Studio Code (VS Code), a lightweight development environment that integrates seamlessly with Azure.

Before you start, ensure that you have the following:

- **Azure subscription**: An active Azure subscription. If you don't have one, you can create a free account.
- **Visual Studio Code**: Download and install Visual Studio Code (https://code.visualstudio.com).
- **Azure Functions extension**: Install the Azure Functions extension for Visual Studio Code (https://marketplace.visualstudio.com/items?itemName=ms-azuretools.vscode-azurefunctions) to help create, manage, and deploy Azure Functions directly from VS Code.
- **Azure Functions Core Tools**: Install the Azure Functions Core Tools (https://learn.microsoft.com/en-us/azure/azure-functions/functions-run-local?tabs=windows%2Cisolated-process%2Cnode-v4%2Cpython-v2%2Chttp-trigger%2Ccontainer-apps&pivots=programming-language-csharp) to run your functions locally for testing.
- **Azurite**: Install the Azurite emulator (https://learn.microsoft.com/en-us/azure/storage/common/storage-use-azurite?tabs=visual-studio-code%2Cblob-storage) for local Azure Storage to run your functions locally for testing.

To illustrate how easy it is to build with Azure Functions, let's create a simple **HTTP-triggered function**.

1. **Install the Azure Functions extension**: After installing Visual Studio Code, open the Extensions view, search for"Azure Functions, and click Install.
2. **Create a new project**: Open the Command Palette in VS Code. Type **Azure Functions: Create New Project** and choose a directory for your project. Select C# as the language and a runtime. Azure Functions supports multiple languages, including C#, JavaScript, Python, Java, and PowerShell.
3. **Select a trigger**: After creating the project, choose a trigger for the function. For this example, select **HTTP Trigger**, one of the most common triggers for APIs and event-driven functions.
4. **Configure the function**: Name the function SimpleHttpFunction and set the access level to Anony-

mous for public access. The function template will generate boilerplate code, typically resembling **Listing 1**.

5. **Test locally**: Ensure that you have Azure Function Core Tools and Azurite emulator installed on your computer. Press F5 in VS Code to run the function locally. Visual Studio Code starts a local server, allowing you to test the function by sending HTTP requests to its URL (e.g., http://localhost:7071/api/SimpleHttpFunction?name=JohnDoe) using tools like Postman or cURL.
6. **Deploy to Azure:** Once satisfied with your local tests, deploy the function to Azure. Ensure that your Azure subscription has a function app that runs your code. Deploy from Visual Studio Code packages and deploy your project to this app. Right-click the function in the Azure extension pane function, select **Deploy to Function App**, and choose your Azure subscription and resource group.

You've successfully created and deployed your first Azure Function following these steps. Next, you'll explore how triggers and bindings enhance the flexibility and functionality of your functions.

Understanding Triggers and Bindings

Triggers and bindings form the backbone of Azure Function' extensibility and flexibility. They dictate how functions are invoked and how they interact with other services.

Triggers initiate function execution. Each function must have one trigger that specifies the type of event that starts the function.

Common trigger types include:

- **HTTP trigger**: Initiates the function via HTTP requests, making it suitable for building APIs and webhooks.
- **Timer trigger**: Runs on a schedule, making it ideal for cron-job-like tasks such as database cleanup or report generation.
- **Blob trigger**: Monitors Azure Storage containers for new or updated blobs, which is great for file processing scenarios.
- **Queue trigger:** Responds to new messages in Azure Storage queues, enabling message-based workflows.

Bindings allow functions to connect seamlessly with other Azure services for input and output operations. Bindings simplify coding by handling the plumbing needed to integrate with various data sources, so you don't have to write extensive connection logic. Types of bindings include the following:

- **Input bindings** allow a function to read data from an external source, like a database or blob storage.
- **Output bindings** allow a function to send data to a service, such as writing to a Cosmos DB collection or sending a message to a queue.

Example binding configuration: In the function.json file, you can define bindings like:

Listing 1: Configure the function

```
using System.IO;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Extensions.Http;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.Logging;
using Newtonsoft.Json;

public static class SimpleHttpFunction
{
    [FunctionName("SimpleHttpFunction")]
    public static async Task<IActionResult> Run(
        [HttpTrigger(AuthorizationLevel.Function,
"get", "post", Route = null)] HttpRequest req,
        ILogger log)
    {
        log.LogInformation(
"C# HTTP trigger function processed a request.");

        string name = req.Query["name"];

        if (string.IsNullOrEmpty(name))
        {
            using (var reader = new StreamReader(req.Body))
            {
                var body = await reader.ReadToEndAsync();
                dynamic data = JsonConvert.DeserializeObject(body);
                name = data?.name;
            }
        }

        return name != null
            ? (ActionResult)new OkObjectResult(
$"Hello from Code Magazine, {name}!")
            : new BadRequestObjectResult(
"Please pass a name on the query string or in the request body.");
    }
}
```

```
{
    "bindings": [
        {
            "type": "httpTrigger",
            "direction": "in",
            "name": "req",
            "methods": ["get", "post"]
        },
        {
            "type": "http",
            "direction": "out",
            "name": "res"
        }
    ]
}
```

Bindings make it easy to enrich your functions without complex configurations or boilerplate code. For instance, you can create a function that processes data from an HTTP request and writes it directly to an Azure Storage table, all within a few lines of code.

Integrating Azure Functions with Other Azure Services

Azure Functions shine when integrated with other Azure services to build comprehensive solutions. Here are some powerful ways to integrate functions with other services:

1. **Azure Logic Apps:** Use Azure Logic Apps to orchestrate workflows that include Azure Functions as

steps. This combination enhances automation and complex workflow creation without extensive code.

2. **Azure Event Grid:** Integrate Azure Functions with **Event Grid** to create event-driven architectures that respond to events from other services such as Blob Storage or custom sources. This approach enables real-time data processing.
3. **Azure Cosmos DB:** Combine Azure Functions with **Cosmos DB bindings** to automatically read and write data to a globally distributed database. This set up allows you to build scalable, data-centric applications.

Listing 2: Error handling in Azure Functions

```
using System;
using System.IO;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Extensions.Http;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.Logging;
using Newtonsoft.Json;

public static class ErrorHandlingExample
{
    [FunctionName("ErrorHandlingExample")]
    public static async Task<IActionResult> Run(
        [HttpTrigger(AuthorizationLevel.Function,
"post", Route = null)] HttpRequest req,
        ILogger log)
    {
        try
        {
          log.LogInformation(
"C# HTTP trigger function processed a request.");

            string requestBody =
await new StreamReader(req.Body).ReadToEndAsync();
            dynamic data = JsonConvert.DeserializeObject(requestBody);

            if (data?.value == null)
            {
                throw new ArgumentNullException("value",
"The 'value' field is required in the request body.");
            }

            int result = ProcessData((int)data.value);
            return new OkObjectResult($"Processed value: {result}");
        }
        catch (ArgumentNullException ex)
        {
            log.LogError($"Input error: {ex.Message}");
            return new BadRequestObjectResult(
$"Missing input: {ex.ParamName}");
        }
        catch (Exception ex)
        {
            log.LogError($"Unexpected error: {ex.Message}");
            return new StatusCodeResult(StatusCodes.Status500InternalServerError);
        }
    }

    private static int ProcessData(int value)
    {
        if (value < 0)
        {
            throw new InvalidOperationException(
"Value must be non-negative.");
        }

        // Simulate data processing
        return value * 2;
    }
}
```

Example: Event-Driven Function with Event Grid

The following Azure Function handles Event Grid events for newly created blobs in Azure Storage. It logs the event details, checks if the event type is BlobCreated, extracts the blob URL, and provides a placeholder for further blob processing.

```
using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Extensions.EventGrid;
using Microsoft.Extensions.Logging;
using Newtonsoft.Json.Linq;

public static class EventGridFunction
{
    [FunctionName("EventGridFunction")]
    public static void Run(
         [EventGridTrigger]
EventGridEvent eventGridEvent,
        ILogger log)
    {
        log.LogInformation(
$"Event received: {eventGridEvent.EventType}");
        var data =
((JObject)eventGridEvent.Data).ToString();
        log.LogInformation($"Event Data: {data}");
    }
}
```

Error Handling and Best Practices

Error handling is crucial in building robust serverless applications. Azure Functions provides several mechanisms for gracefully handling errors to ensure reliability and maintainability. Key practices for error handling include:

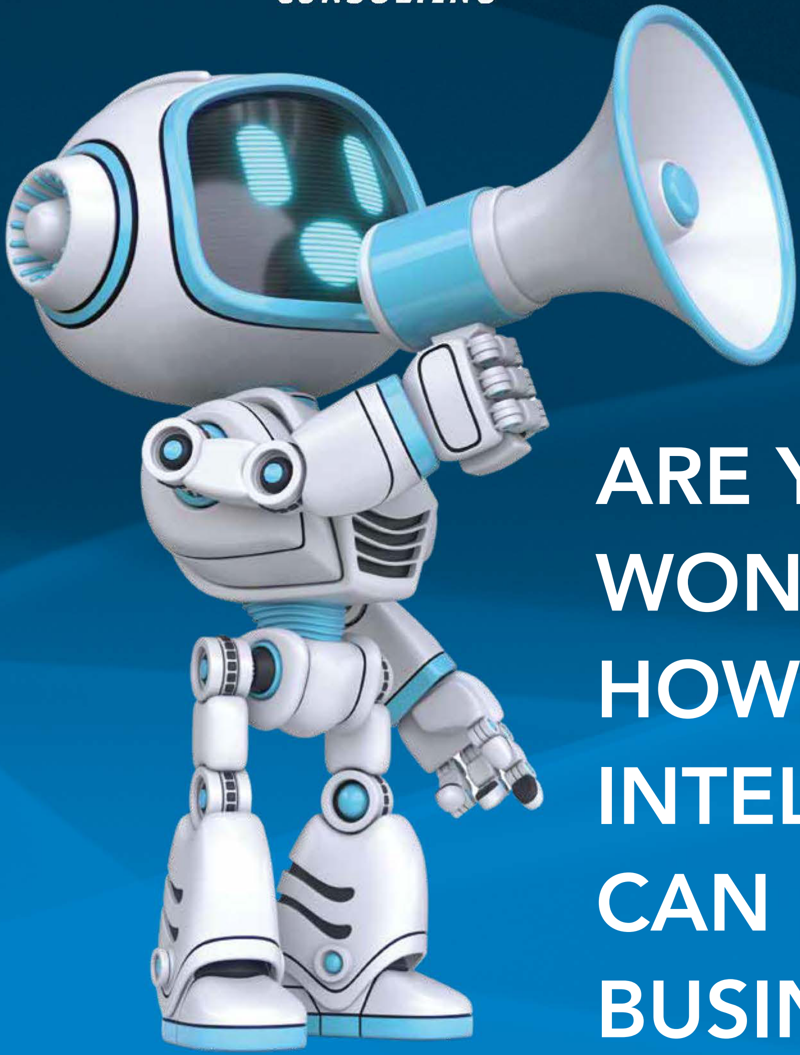
- **Structured exception handling:** Use try-catch blocks in your code to catch exceptions and log meaningful error messages. This helps diagnose issues quickly.
- **Retry policies:** Configure retry policies for triggers like **Queue** and **Event Hub** to automatically reprocess failed executions.
- **Custom logging:** Use context.log for custom log entries and integrate with **Application Insights** for advanced monitoring and error tracking.
- **Dead-lettering:** For functions triggered by queues, enable dead-lettering to capture messages that fail after multiple retries for further analysis.

Example: Enhanced Error Handling

In **Listing 2**, if the input is invalid or an error occurs, Azure function logs the error and returns an error response with status 500.

Accessing Azure SQL Database and Returning Data via API

Let's dive into an example where an Azure Function retrieves data from an **Azure SQL Database** and exposes it through an HTTP API. This is a common use case for building lightweight APIs without the need for a full-fledged web server.



CODE
CONSULTING

ARE YOU WONDERING HOW ARTIFICIAL INTELLIGENCE CAN HELP YOUR BUSINESS?

Do you worry about privacy or regulatory issues stopping you from using AI to its fullest?

We have the answers!

We will send an expert to your office to meet with you. You will receive:

1. An overview presentation of the current state of Artificial Intelligence.
2. How to use AI in your business while ensuring privacy of your and your clients' information.
3. A sample application built on your own HR documents – allowing your employees to query those documents in English and cutting down the number of questions that you and your HR group have to answer.
4. A roadmap for future use of AI catered to what you do.

CONTACT US TODAY FOR A FREE CONSULTATION AND DETAILS ABOUT OUR SERVICES.

codemag.com/executivebriefing
832-717-4445 ext. 9 • info@codemag.com

©shutterstock

Step 1: Set Up the Azure SQL Database

Before creating the function, ensure that you have:

- An Azure SQL Database set up.
- A table with sample data.
- Connection details for the database.

Listing 3: Azure Function that retrieves data from an Azure SQL Database and exposes it through an HTTP API

```
using System;
using System.Data.SqlClient;
using System.IO;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Extensions.Http;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.Logging;
using Newtonsoft.Json;
using System.Collections.Generic;

public static class GetProductsFunction
{
    [FunctionName("GetProducts")]
    public static async Task<IActionResult> Run(
        [HttpTrigger(AuthorizationLevel.Function, "get",
Route = "products")] HttpRequest req, ILogger log)
    {
        string connectionString = Environment.GetEnvironmentVariable("SqlConnectionString");

        var products = new List<Product>();
        try
        {
            using (SqlConnection conn =
new SqlConnection(connectionString))
            {
                conn.Open();
                var query = "SELECT Id, Name, Price FROM Products";
                using (SqlCommand cmd = new SqlCommand(query, conn))
                {
                    using (SqlDataReader reader =
await cmd.ExecuteReaderAsync())
                    {
                        while (reader.Read())
                        {
                            products.Add(new Product
                            {
                                Id = reader.GetInt32(0),
                                Name = reader.GetString(1),
                                Price = reader.GetDecimal(2)
                            });
                        }
                    }
                }
            }

            return new OkObjectResult(products);
        }
        catch (Exception ex)
        {
            log.LogError($"Error fetching products: {ex.Message}");
            return new StatusCodeResult(StatusCodes.Status500InternalServerError);
        }
    }

    public class Product
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public decimal Price { get; set; }
    }
}
```

For demonstration, let's assume a database table called Products:

```
CREATE TABLE Products (
    Id INT PRIMARY KEY,
    Name NVARCHAR(100),
    Price DECIMAL(10, 2)
);

INSERT INTO Products (Id, Name, Price) VALUES
(1, 'Laptop', 899.99),
(2, 'Smartphone', 699.99),
(3, 'Tablet', 299.99);
```

Step 2: Create the Azure Function

Listing-3 shows the code for Azure Function that retrieves data from an **Azure SQL Database** and exposes it through an HTTP API. Key points to note about this function:

- The function retrieves data from the Products table in Azure SQL.
- The connection string is retrieved from environment variables (SqlConnectionString).

Step 3: Set the Connection String in Azure

In the Azure Portal:

1. Navigate to your Function App.
2. Under **Configuration**, add a new application setting:
    - **Name**: SqlConnectionString
    - **Value**: Connection string for your Azure SQL Database

Step 4: Consume the Azure Function in a Console App

Here's a C# Console Application to call the Azure Function and display the data, as shown in **Listing 4**.

Key points about consuming the function:

- Replace <your-function-app-name> with the name of your Azure Function App.
- Use <your-function-key> if your function requires a key for authorization.
- The console app fetches data from the API and displays it.

This example demonstrates how Azure Functions can be used to expose data from an Azure SQL Database as a lightweight API. Following this pattern, you can create scalable APIs that serve data without managing a dedicated web server. The provided console app shows how to consume this API, making it easier to integrate with various clients.

Understanding Azure Functions Pricing

Azure Functions offers flexible pricing models to accommodate various application needs. Understanding these models helps optimize costs and align them with your project's requirements.

- **Consumption Plan:** In the Consumption Plan, you pay only for the resources your functions consume. Billing is based on the number of executions and the compute resources used, measured in gigabyte-seconds (GB-s). This plan includes a monthly free grant of 1 million requests and 400,000 GB-s of resource consumption per subscription.

- **Premium Plan**: The Premium Plan offers enhanced performance, including features like virtual network connectivity and no cold starts. Billing is based on the number of core seconds and memory allocated across instances. Unlike the Consumption Plan, there is no execution charge, but at least one instance must be allocated at all times per plan.
- **Flex Consumption Plan (Preview)**: The Flex Consumption Plan provides high scalability with compute choices and virtual networking. It operates in two modes:
  - **On Demand**: Instances scale based on configured per-instance concurrency, with billing only for the time instances are executing functions. This mode includes a monthly free grant of 250,000 executions and 100,000 GB-s of resource consumption per subscription.
  - **Always Ready**: You can configure instances to be always enabled and assigned to specific triggers and functions. Billing includes the total amount of memory provisioned for baseline and execution time, as well as the total number of executions. There are no free grants in this mode.
- **Dedicated (App Service) Plan:** In the Dedicated Plan, functions run on dedicated virtual machines under an App Service Plan. Billing follows standard App Service Plan rates, making it suitable for scenarios requiring predictable costs and dedicated resources.

Tips for cost management:

- **Monitor usage**: Use Azure Monitor and Application Insights to track function executions and resource consumption.
- **Choose the right plan**: Select a plan that aligns with your application's performance and scaling needs.
- **Leverage free grants**: Use the monthly free grants in the Consumption and Flex Consumption Plans to minimize costs.

By understanding these pricing models and selecting the appropriate plan, you can effectively manage costs while leveraging the full capabilities of Azure Functions.

Performance Optimization Tips

Optimizing performance ensures that your Azure Functions scale effectively and respond to user demands. Here are key optimization strategies:

- **Cold start minimization:** Functions may experience a cold start when running in the Consumption Plan. To minimize cold starts:
  - Use the Premium Plan or Dedicated Plan, which keeps instances warm.
  - Keep functions lightweight, minimizing initialization logic.
- **Efficient code practices:** Follow coding best practices to use Azure functions efficiently.
  - Avoid long-running tasks. Functions should complete quickly; use durable functions for workflows that need to persist state.
  - Use async programming. Write non-blocking code to improve response times.
  - Reduce dependencies. Only import necessary modules to decrease startup time.
- **Monitor and analyze performance:** Integrate **Application Insights** to monitor performance metrics, track failures, and analyze execution times. Use this data to identify and resolve bottlenecks.

Listing 4: Console app calling Azure Function

```
using System;
using System.Net.Http;
using System.Threading.Tasks;
using System.Collections.Generic;
using Newtonsoft.Json;

class Program
{
    static async Task Main(string[] args)
    {
        string functionUrl =
"https://<your-function-app-name>.azurewebsites.net/api/products";
        string functionKey =
"<your-function-key>"; // If required, add the Function key

        using (HttpClient client = new HttpClient())
        {
            client.DefaultRequestHeaders.Add(
"x-functions-key", functionKey); // Add key if needed
            HttpResponseMessage response =
await client.GetAsync(functionUrl);

            if (response.IsSuccessStatusCode)
            {
                string responseData =
await response.Content.ReadAsStringAsync();
                var products =
JsonConvert.DeserializeObject<List<Product>>(responseData);
                foreach (var product in products)
                {
                    Console.WriteLine($"ID: {product.Id},
Name: {product.Name}, Price: {product.Price:C}");
                }
            }
            else
            {
                Console.WriteLine($"Error: {response.StatusCode}");
            }
        }
    }

    public class Product
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public decimal Price { get; set; }
    }
}
```

Security Best Practices for Azure Functions

Security is critical for any application. Here are some best practices:

1. **Managed identities**: Enable managed identities to access Azure resources without embedding credentials in code securely.
2. **Enforce HTTPS**: Ensure that all traffic to your function app uses HTTPS for secure data transmission.

SPONSORED SIDEBAR:

Add Copilots to Your Apps

The future is here now and you don't want to get left behind. Unlock the **true potential** of your software applications by adding **Copilots**.

CODE Consulting can assess your applications and provide you with a roadmap for adding Copilot features and optionally assist you in adding them to your applications.

Reach out to us today to get your application assessment scheduled. www.codemag.com/ai

3. **Implement role-based access control (RBAC)**: Limit access to your Azure Functions using RBAC to assign the minimum necessary permissions.
4. **Data encryption**: Use **Azure Key Vault** to manage and access secrets, keys, and certificates securely.
5. **Monitoring**: Integrate **Application Insights** for real-time monitoring, logging, and alerting to detect anomalies and track security events.

Advanced Use Cases for Azure Functions

Azure Functions provide immense flexibility, making them suitable for more complex and innovative applications. Beyond basic functions, developers can leverage the power of Azure Functions to build real-time, scalable, and highly integrated solutions. Below are some advanced use cases that showcase the versatility of Azure Functions:

- **Real-time data processing**: Pair with **Event Hub** or **IoT Hub** to handle real-time high-velocity data streams. This set-up is ideal for real-time monitoring, live data analytics, or IoT data ingestion.
- **Microservices architecture**: Build serverless microservices that communicate using **Azure Service Bus** or **Event Grid**. This approach promotes modularity, allowing you to create loosely coupled services that can scale independently.
- **Workflow automation**: Automate business processes triggered by changes in data or application events. Azure Functions can act as workflow steps, streamlining processes like document approvals, report generation, or CRM updates.
- **Integration with third-party services**: Create Azure Functions that connect with external APIs and services to extend application capabilities. This use case supports building middleware for payment processing, customer relationship management (CRM) integrations, or social media interactions.

These advanced scenarios illustrate Azure Functions' extensive capabilities and how they can be leveraged to build complex, cloud-native solutions.

Conclusion

Azure Functions offers an accessible, flexible, and scalable way to build serverless applications that respond to various triggers and integrate seamlessly with other Azure services. By following best practices, implementing robust error handling, optimizing performance, and ensuring security, developers can create robust solutions ready for production. Dive into serverless development and unlock the potential of building modern, cloud-native applications.

Vaibhav Gujral

CODE COMPILERS

CODE

May/Jun 2025
Volume 26 Issue 3

Group Publisher
Markus Egger

Editor-in-Chief
Rod Paddock

Managing Editor
Ellen Whitney

Content Editor
Melanie Spiller

Writers in This Issue
Vaibhav Gujral
Joydip Kanjilal
Paul D. Sheriff
Matthew Hess
Sahil Malik

Technical Reviewers
Markus Egger
Rod Paddock

Production
Friedl Raffeiner Grafik Studio
www.frigraf.it

Graphic Layout
Friedl Raffeiner Grafik Studio in collaboration
with onsight (www.onsightdesign.info)

Printing
Fry Communications, Inc.
800 West Church Rd.
Mechanicsburg, PA 17055

Advertising Sales
Tammy Ferguson
832-717-4445 ext. 26
tammy@code-magazine.com

Circulation & Distribution
General Circulation: EPS Software Corp.
Newsstand: The Mail Group (TMG)
Accelerate 360

Subscriptions

Circulation Manager
Colleen Cade
832-717-4445 ext. 28
ccade@codemag.com

US subscriptions are $29.99 USD for one year.
Subscriptions outside the US are $50.99 USD.
Payments should be made in US dollars drawn
on a US bank. American Express, MasterCard,
Visa and Discover credit cards accepted.
Back issues are available. For subscription
information, email subscriptions@code-magazine.com
or contact customer service at 832-717-4445 ext. 9.

Subscribe online at
www.code-magazine.com
CODE Developer Magazine
EPS Software Corporation / Publishing Division
6605 Cypresswood Drive, Ste 425, Spring, Texas 77379 USA
Phone: 832-717-4445

dtSearch

IRONSOFTWARE

CODE STAFFING

next GenAI CONFERENCE

YOUR PARTNER FOR CUSTOM SOFTWARE SOLUTIONS

CODE

REAL BUSINESS VALUE FOR AI

TRAINING / MENTORING

CUSTOM APPLICATION DEVELOPMENT

CONTINGENT IT STAFFING

Is your development team struggling to complete business-critical projects on time? Are you looking to harness cutting-edge technologies, including AI, for maximum impact?

CODE EXCELS IN:

- **AI integration for enhanced functionality**
- **.NET web and desktop development**
- **Azure cloud migration and transformation**
- **Blazor development**
- **Mobile app creation**
- **Staffing, training and mentoring**

Let CODE transform your software challenges into competitive advantages.

CONTACT US TODAY. NO STRINGS. NO COMMITMENT.

codemag.com/code

832-717-4445 ext. 9 • info@codemag.com

©shutterstock

CODE
STAFFING

UNLOCK STAFFING EXCELLENCE

Top-Notch IT Talent, Contract Flexibility, Happy Teams, and a Commitment to Customer Success Converge with CODE Staffing

Our IT staffing solutions are engineered to drive your business forward while saving you time and money. Say goodbye to excessive overhead costs and lengthy recruitment efforts. With CODE Staffing, you'll benefit from contract flexibility that caters to both project-based and permanent placements. We optimize your workforce strategy, ensuring a perfect fit for every role and helping you achieve continued operational excellence.

Ready to Discuss Your IT Staffing Needs?

Visit our website to find out more about how we are changing the staffing industry.

Website: codestaffing.com
Yair Alan Griver (yag)
Chief Executive Officer
Direct: +1 425 301 1590
Email: yag@codestaffing.com